Electronic Notes in Theoretical Computer Science 89 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 16 pages

Automated Compositional Abstraction Refinement for Concurrent C Programs: A Two-Level Approach [1]

Sagar Chaki Joël Ouaknine Karen Yorav Edmund Clarke

Computer Science Department
Carnegie Mellon University
Pittsburgh PA 15213, USA
Email: {chaki|ouaknine|kareny|emc}@cs.cmu.edu

Abstract

The state space explosion problem in model checking remains the chief obstacle to the practical verification of real-world distributed systems. We attempt to address this problem in the context of verifying concurrent (message-passing) C programs against safety specifications. More specifically, we present a fully automated compositional framework which combines two orthogonal abstraction techniques (operating respectively on data and events) within a counterexample-guided abstraction refinement (CEGAR) scheme. In this way, our algorithm incrementally increases the granularity of the abstractions until the specification is either established or refuted. Our explicit use of compositionality delays the onset of state space explosion for as long as possible. To our knowledge, this is the first compositional use of CEGAR in the context of model checking concurrent C programs. We describe our approach in detail, and report on some very encouraging preliminary experimental results obtained with our tool MAGIC.

1 Introduction

Formal verification of distributed software has long been acknowledged to be a difficult yet important task. For this reason, there has been a tremendous

[1] 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, and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485. The views and conclusions contained in this document are those of the author 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.

amount of research over the years devoted to the abstract modelling and validation of concurrent systems and their specifications. Many paradigms and techniques, ranging from process algebra and model checking to predicate abstraction and counterexample-guided abstraction refinement (CEGAR), have been proposed towards the ultimate goal of automatically verifying large distributed applications written in industry-level programming languages.

The majority of these advances target specific—and often orthogonal—aspects of the problem, but fail to solve it as a whole. The work we present here combines several of these techniques to efficiently verify global specifications on concurrent C programs in a fully automated way. More specifically, we focus on reactive systems, implemented using concurrent C programs that communicate with each other through synchronous (blocking) message-passing. Examples of such systems include client-server protocols, schedulers, telecommunication applications, etc. We consider safety specifications, in other words requirements describing the sequences of messages (or events) that the system is allowed to produce, or equivalently the 'bad' states that the system is meant to avoid.

We propose a fully automated *compositional two-level counterexample-guided abstraction refinement* scheme to verify that a parallel composition $C_1||\ldots||C_n$ of n sequential C programs satisfies a specification *Spec*. We first use predicate abstraction to transform conservatively (insofar as *safety properties* are concerned) each (infinite-state) C program C_i into a finite-state process P_i. Since the parallel composition of these P_i's may well still have an unmanageably large state space, we further reduce each P_i by conservatively aggregating states together, based on the events they can perform, yielding a smaller process A_i; only then do we explicitly build the global state space of the much coarser parallel composition $A = A_1||\ldots||A_n$. By construction, A exhibits all of the original system's behaviours, and usually many more. We then check A against the specification *Spec*. If successful, we conclude that our original system $C_1||\ldots||C_n$ is safe. Otherwise, we must examine the counterexample obtained to determine whether it is valid (at the lower levels) or not. It is important to note that this validation can be carried out level- and component-wise, without it ever being necessary to construct in full the large state space of the whole system. A valid counterexample at the lowest level shows *Spec* to be violated and thus terminates the procedure. Otherwise, a (component-specific) refinement of the appropriate abstracted system is carried out, eliminating the spurious counterexample, and the algorithm proceeds with a new iteration of the verification cycle.

The crucial features of our approach therefore consist of the following:

- We leverage two very different kinds of abstraction to reduce a parallel composition of sequential C programs to a very coarse parallel composition of finite-state processes. The first (predicate) abstraction partitions the (potentially infinite) state space according to the possible values of variables, whereas the second abstraction lumps these resulting states together according to the events that they can communicate.

- A counterexample-guided abstraction refinement scheme incrementally refines these abstractions until the right granularity is achieved to decide whether the specification holds or not. We note that while termination of the entire algorithm obviously cannot be guaranteed [2], all of our experimental examples could be handled without requiring human input.
- Our use of compositional reasoning, grounded in standard process algebraic techniques, enables us to perform most of our analysis component by component, without ever having to construct global state spaces except at the highest (most abstract) level.

The verification procedure is fully automated, and requires no user input beyond supplying the C programs and the specification to be verified. We have implemented the algorithm within our tool MAGIC (**M**odular **A**nalysis of pro**G**rams **I**n **C**) [2,9] and have carried out a number of case studies, which we report here. To our knowledge, our algorithm is the first to invoke CEGAR over more than a single abstraction refinement scheme (and in particular over *action-based* abstractions), and also the first to combine CEGAR with fully automatic compositional reasoning for concurrent systems.

The experiments we have carried out range over a variety of sequential and concurrent examples, and yield promising results. With the smaller examples we find that our two-level approach constructs models that are 2 to 11 times smaller than those generated by predicate abstraction alone. These ratios increase dramatically as we consider larger and larger examples. In some of our instances MAGIC constructs models that are more than two orders of magnitude smaller than those created by mere predicate abstraction. Full details are presented in Section 5.

Foundations and Related Work

Predicate abstraction was introduced in [37] as a means to transform conservatively infinite-state systems into finite-state ones, so as to enable the use of finitary techniques such as model checking [12,11]. It has since been widely used—see, for instance [17,21,18,32,5,20]. The technique we employ to generate automatically suitable predicates is described in [9].

The formalization of the more general notion of abstraction first appeared in [19]. We distinguish between *exact* abstractions, which preserve all properties of interest of the system, and *conservative* abstractions—used in this paper—which are only guaranteed to preserve 'undesirable' properties of the system (e.g., [27,14]). The advantage of the latter is that they usually lead to much greater reductions in the state space than their exact counterparts. However, conservative abstractions in general require an iterated abstraction refinement mechanism (such as CEGAR [13]) in order to establish specification

[2] This of course follows from the fact that the halting problem is undecidable.

satisfaction.

The abstractions we use on finite-state processes essentially lump together states that can perform the same set of actions, and gradually refine these partitions according to reachable successor states. Our refinement procedure can be seen as an atomic step of the Paige-Tarjan algorithm [34], and therefore yields successive abstractions which converge in a finite number of steps to the bisimulation quotient of the original process.

Counterexample-guided abstraction refinement [13,28], 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 [33] (in non-automated form), and [6,35,29,24,10,15].

Compositionality, which features crucially in our work, is broadly concerned with the preservation of properties under substitution of components in concurrent systems. It has been most extensively studied in process algebra (e.g., [26,31,36]), particularly in conjunction with abstraction. In [7], 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.

A technique closely related to compositionality is that of assume-guarantee reasoning [22,30,25]. It was originally developed to circumvent the difficulties associated with generating exact abstractions, and has recently been implemented as part of a fully automated and incremental verification framework [16].

Among the works most closely resembling ours we note the following. The Bandera project [18] offers tool support for the automated verification of Java programs based on abstract interpretation; there is no automated CEGAR and no explicit compositional support for concurrency. [35] imports Bandera-derived abstractions into an extension of Java PathFinder which incorporates CEGAR. However, once again no use is made of compositionality, and only a single level of abstraction is considered. [38] describes another tool implemented in Java PathFinder which explicitly supports concurrency; it uses datatype abstraction on the first level, and partial order reduction with aggregation of invisible transitions on the second level. Since all abstractions are exact it does not require the use of CEGAR. The SLAM project [3,6,5] has been very successful in analyzing interfaces written in C. It is built around a single-level predicate abstraction and automated CEGAR treatment, and offers no explicit compositional support for concurrency. Lastly, the BLAST project [1,24,23] proposes a single-level lazy (on-the-fly) predicate abstraction scheme together with CEGAR and thread-modular assume-guarantee reasoning. The BLAST framework is based on shared variables rather than message-passing as the communication mechanism.

The next section presents a series of standard definitions that are used throughout the paper. Section 3 then describes the two-level CEGAR algorithm, while Section 4 presents our action-guided CEGAR procedure. Section 5 summarizes the results of our experiments. Finally, Section 6 offers conclusions and avenues for future work.

2 Preliminaries

A labelled transition system (LTS for short) is a quadruple $\langle S, init, Act, T\rangle$ with S a finite set of states, $init \in S$ an initial state, Act a finite set (alphabet) of actions (or events), and $T \subseteq S \times A \times S$ a transition relation. We often write $s \xrightarrow{a} t$ to mean $(s, a, t) \in T$. In this section, unless noted otherwise, we assume a fixed LTS $M = \langle S, init, Act, T\rangle$.

A trace π is a finite (possibly empty) sequence of actions. We define the language $L(M)$ of the LTS M to be the set of all traces $a_1 \dots a_n \in Act^*$ such that, for some sequence $s_0 \dots s_n$ of states of M (with $s_0 = init$) we have $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$. We refer to the underlying sequence of states $s_0 \dots s_n$ as the path in M corresponding to the trace $a_1 \dots a_n$.

For $s \in S$ we write $enabled(s) = \{a \in Act \mid \exists t \in S \centerdot s \xrightarrow{a} t\}$ to denote the set of actions enabled in state s.

For a trace $\pi = a_1 \dots a_n \in Act^*$ and $s, t \in S$ two states of M, we write $s \stackrel{\pi}{\Longrightarrow} t$ to indicate that t is reachable from s through π, i.e., that there exist states $s_0 \dots s_n$ with $s = s_0$ and $t = s_n$, such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$. Given a state $s \in S$ and a trace $\pi \in Act^*$, let $Reach(M, s, \pi) = \{t \in S | s \stackrel{\pi}{\Longrightarrow} t\}$ stand for the set of states reachable from s through π. We overload this notation by setting, for a set of states $Q \subseteq S$, $Reach(M, Q, \pi) = \{t \in S \mid \exists q \in Q \centerdot q \stackrel{\pi}{\Longrightarrow} t\}$; this represents the set of states reachable through π from some state in Q.

Let $\pi \in Act^*$ be a trace over Act, and let Act' be another (not necessarily disjoint) set of actions. The projection $\pi\restriction_{Act'}$ of π on Act' is the subtrace of π obtained by simply removing all actions in π that are not in Act'.

Let $M_1 = \langle S_1, init_1, Act_1, T_1\rangle$ and $M_2 = \langle S_2, init_2, Act_2, T_2\rangle$ be two LTSs. Their parallel composition $M_1||M_2 = \langle S_1 \times S_2, (init_1, init_2), Act_1 \cup Act_2, T_1||T_2\rangle$ is defined so that $((s_1, s_2), a, (t_1, t_2)) \in T_1||T_2$ iff one of the following holds:

(i) $a \in Act_1 \setminus Act_2$ and $s_1 \xrightarrow{a} t_1$ and $s_2 = t_2$.

(ii) $a \in Act_2 \setminus Act_1$ and $s_2 \xrightarrow{a} t_2$ and $s_1 = t_1$.

(iii) $a \in Act_1 \cap Act_2$ and $s_1 \xrightarrow{a} t_1$ and $s_2 \xrightarrow{a} t_2$.

In other words, components must synchronize on shared actions and proceed independently on local actions. This notion of parallel composition has been used in, e.g., CSP [26], and in the work of Anantharaman et al. [4]. We refer the reader to [36] for proofs of the following standard results:

Theorem 2.1

(i) *Parallel composition is associative and commutative as far as the accepted language is concerned. Thus, in particular, no bracketing is required when combining more than two LTSs.*

(ii) *Let $M_1, \ldots, M_n$ and $M'_1, \ldots, M'_n$ be LTSs with every pair of LTSs M_i, M'_i sharing the same alphabet $Act_i = Act'_i$. If, for each $1 \leqslant i \leqslant n$, we have $L(M_i) \subseteq L(M'_i)$, then $L(M_1||\ldots||M_n) \subseteq L(M'_1||\ldots||M'_n)$. In other words, parallel composition preserves language containment.*

(iii) *Let $M_1, \ldots, M_n$ be LTSs with respective alphabets $Act_1, \ldots, Act_n$, and let π be any trace. Then $\pi \in L(M_1||\ldots||M_n)$ iff, for each $1 \leqslant i \leqslant n$, we have $\pi\lceil_{Act_i} \in L(M_i)$. In other words, whether a trace belongs to a parallel composition of LTSs can be checked by projecting and examining the trace on each individual component separately.*

Theorem 2.1 forms the basis of our compositional approach to verification.

We consider a concurrent version of the C programming language in which a fixed number of sequential programs $C_1, \ldots, C_n$ are run concurrently on independent platforms. Each program C_i has an associated alphabet of actions Act_i, and can communicate a particular event a in its alphabet only if all other programs having a in their alphabets are willing to synchronize on this event. An action is realized in C using a call to a library routine. Programs have local variables but no shared variables. In other words, we are assuming blocking message-passing (i.e., 'send' and 'receive' statements) as the sole communication mechanism. Given such a parallel composition $C_1||\ldots||C_n$ of C programs, we write $L(C_1||\ldots||C_n)$ to denote the set of all possible traces of events which $C_1||\ldots||C_n$ can communicate. At present, the full syntax of ANSI C is supported, with the exception of pointers, recursion, and floating-point arithmetic. We refer the reader to [9] for more details.

Our goal is to verify that the concurrent C program $C_1||\ldots||C_n$ satisfies a specification *Spec*, where the latter is expressed as an LTS. We use trace containment as our notion of conformance: the concurrent program meets its specification iff $L(C_1||\ldots||C_n) \subseteq L(Spec)$.

3 Two-Level Counterexample-Guided Abstraction Refinement

Consider a concurrent C program $C_1||\ldots||C_n$ and a specification *Spec*. We first invoke predicate abstraction to reduce each (infinite-state) program C_i into a finite LTS (or process) P_i having the same alphabet as C_i. The initial abstraction is created with a relatively small set of predicates, and further predicates are then added as required to refine the P_i's and eliminate spurious counterexamples. This procedure may add a large number of predicates, yielding an abstract model with a potentially huge state space. We therefore

```
Input: C programs C_1,...,C_n and specification Spec
Output: 'C_1||...||C_n satisfies Spec' or
        counterexample π ∈ L(C_1||...||C_n) \ L(Spec)

  predicate abst.:  create LTSs P_1,...,P_n with L(C_i) ⊆ L(P_i)
† action-guided abst.:  create LTSs A_1,...,A_n with L(P_i) ⊆ L(A_i)
  repeat
    if L(A_1||...||A_n) ⊆ L(Spec) return 'C_1||...||C_n satisfies Spec'
    else
      extract counterexample π ∈ L(A_1||...||A_n) \ L(Spec)
      if π ∈ L(P_1||...||P_n)
        if π ∈ L(C_1||...||C_n) return π
        else
          do predicate abstraction refinement of P_1,...,P_n
†         adjust or create new abstractions A_1,...,A_n
      else
‡       do action-guided refinement of A_1,...,A_n to eliminate π
  endrepeat.
```

Fig. 1. Two-level CEGAR algorithm.

seek to further reduce each P_i into an LTS A_i with fewer states, again having the same alphabet as C_i. Both abstractions are such that they maintain the language containment $L(C_i) \subseteq L(P_i) \subseteq L(A_i)$. Theorem 2.1 then immediately yields the rule:

$$L(A_1||\dots||A_n) \subseteq L(Spec) \Rightarrow L(C_1||\dots||C_n) \subseteq L(Spec)$$

The converse need not hold: it is possible for a trace $\pi \notin Spec$ to belong to $L(A_1||\dots||A_n)$ but not to $L(C_1||\dots||C_n)$. Such a spurious counterexample is then eliminated, either by suitably refining the A_i's (if $\pi \notin L(P_1||\dots||P_n)$), or by refining the P_i's (and subsequently adjusting the A_i's to reflect this change). The chief property of our refinement procedure (whether at the A_i or the P_i level) is that it purges the spurious counterexample by restricting the accepted language yet maintains the invariant $L(C_i) \subseteq L(P_i') \subset L(P_i) \subseteq L(A_i') \subset L(A_i)$, where primed terms denote refined processes. Note that, according to Theorem 2.1, we can check whether $\pi \in L(P_1||\dots||P_n)$ and whether $\pi \in L(C_1||\dots||C_n)$ one sequential component at a time, without it ever being necessary to construct the full state spaces of the parallel compositions. This iterated process forms the basis of our two-level CEGAR algorithm.

We describe this algorithm in Figure 1. The predicate abstraction and refinement procedure is detailed in [9]. We present our *action-guided* abstraction and refinement steps (marked † and ‡ respectively) in Section 4.

4 Action-Guided Abstraction

We present a CEGAR scheme that operates on LTSs. Given an LTS $P = \langle S, init, Act, T \rangle$, we first create an LTS $A^0 = \langle S^0_A, init^0_A, Act, T^0_A \rangle$ such that (i) $L(P) \subseteq L(A^0)$ and (ii) A^0 contains at most as many states as P (and typically many fewer). Given an abstraction $A = \langle S_A, init_A, Act, T_A \rangle$ of P and a trace $\pi \in L(A) \setminus L(P)$, our refinement procedure produces a refined abstraction $A' = \langle S'_A, init'_A, Act, T'_A \rangle$ such that (i) $L(P) \subseteq L(A') \subset L(A)$, (ii) $\pi \notin L(A')$, and (iii) A' contains at most as many states as P. It is important to note that we require throughout that P, A^0, A, and A' all share the same alphabet. We also remark that iterating this refinement procedure must converge in a finite number of steps to an LTS that accepts the same language as P.

Let us write $B = \langle S_B, init_B, Act, T_B \rangle$ to denote a generic abstraction of P. States of B are called *abstract* states, whereas states of P are called *concrete* states. In our framework, abstract states are always disjoint sets of concrete states that partition S, and our abstraction refinement step corresponds precisely to a refinement of the partition. For $s \in S$ a concrete state, the unique abstract state of B to which s belongs is written $[s]_B$.

In any abstraction B that we generate, a partition S_B of the concrete states of P uniquely determines the abstract model B: the initial state $init_B$ of B is simply $[init]_B$, and for any pair of abstract states $u, v \in S_B$ and any action $a \in Act$, we postulate a transition $u \xrightarrow{a} v \in T_B$ iff there exist concrete states $s \in u$ and $t \in v$ such that $s \xrightarrow{a} t$. This construction is an instance of an *existential abstraction* [14]. It is straightforward to show that it is sound, i.e., that $L(P) \subseteq L(B)$ always holds.

The initial partition S^0_A of concrete states identifies two states $s, t \in S$ if they share the same set of immediately enabled actions: $t \in [s]^0_A$ iff $enabled(t) = enabled(s)$. We then let $S^0_A = \{[s]^0_A \,|\, s \in S\}$. Again, this uniquely defines our initial abstraction A^0, the construction marked † on Figure 1.

In order to describe the refinement step, we need an auxiliary definition. Given an abstract state $u \in S_B$ and an action $a \in Act$, we construct a refined partition $S'_B = Split(S_B, u, a)$ of S which agrees with S_B outside of u, but distinguishes concrete states in u if they have different abstract a-successors in S_B. More precisely, for any $s \in S$, if $s \notin u$, we let $[s]_{B'} = [s]_B$. Otherwise, for $s, t \in u$, we let $[s]_{B'} = [t]_{B'}$ iff $\bigcup\{[s']_B \,|\, s' \in Reach(P, s, a)\} = \bigcup\{[t']_B \,|\, t' \in Reach(P, t, a)\}$. We then let $Split(S_B, u, a) = \{[s]_{B'} \,|\, s \in S\}$. This refined partition uniquely defines a new abstraction, which we write $Abs(Split(S_B, u, a))$. Note that in order to compute the transition relation of $Abs(Split(S_B, u, a))$ it suffices to adjust only those transitions in T_B that have u either as source or target.

The refinement step takes as input a 'spurious' trace $\pi \in L(A) \setminus L(P)$ and returns a refined abstraction A' which does not accept π. This is achieved by repeatedly splitting states of A along abstract paths which accept π. The

Input: `abstraction` A `of` P `(with` $L(P) \subseteq L(A)$`) and`
`trace` $\pi = a_1 \dots a_m \in L(A) \setminus L(P)$
Output: `refined abstraction` A' `(with` $L(P) \subseteq L(A') \subset L(A)$`) and`
$\pi \notin L(A')$

while `there exists some abstract path` $u_0 \xrightarrow{a_1} \dots \xrightarrow{a_m} u_m$ `in` A `do`
  let $\mathit{reachable\_states} = \{\mathit{init}\}$ `/*` init `= initial state of` P `*/`
  let $j = 1$
  while $\mathit{reachable\_states} \neq \emptyset$ **do**
    let $\mathit{reachable\_states} = \mathit{Reach}(P, \mathit{reachable\_states}, a_j) \cap u_j$
    let $j = j + 1$
  endwhile
  let $A = \mathit{Abs}(\mathit{Split}(S_A, u_{j-2}, a_{j-1}))$ `/*` S_A `= set of states of` A `*/`
endwhile
let $A' = A$
return A'.

Fig. 2. Action-guided CEGAR algorithm on LTS.

algorithm in Figure 2 (marked ‡ in Figure 1) describes this procedure in detail.

Theorem 4.1 *The algorithm described in Figure 2 is correct and always terminates.*

Proof. We first note that it is immediate that whenever the algorithm terminates it does return an abstraction A' with $\pi \notin L(A')$. It is equally clear, since A' is obtained via successive refinements of A, that $L(P) \subseteq L(A') \subset L(A)$. It remains to show that every splitting operation performed by the algorithm results in a proper partition refinement; termination then follows from the fact that the set of states of P is finite.

Observe that, since $\pi \notin L(P)$, $\mathit{Reach}(P, \mathit{init}, \pi) = \emptyset$, and therefore the inner while loop always terminates. At that point, we claim that (i) there is an abstract transition $u_{j-2} \xrightarrow{a_{j-1}} u_{j-1}$; (ii) there are some concrete states in u_{j-2} reachable (in P) from *init*; and (iii) none of these reachable concrete states have concrete a_{j-1}-successors in u_{j-1}. Note that (ii) follows from the fact that the inner loop is entered with $\mathit{reachable\_states} = \{\mathit{init}\}$, whereas (i) and (iii) are immediate. Because of the existential definition of the abstract transition relation, we conclude that u_{j-2} contains two kinds of concrete states: some having concrete a_{j-1}-successors in u_{j-1}, and some not. Splitting state u_{j-2} according to action a_{j-1} therefore produces a proper refinement. □

We remark again that each splitting operation is similar to a unit step of the Paige-Tarjan algorithm [34]. Iterating our refinement procedure therefore converges to the bisimulation quotient of P.

We stress that the CEGAR algorithm described in Figure 1 never invokes the above abstraction refinement routine with the full parallel composition

$A = A_1||\ldots||A_n$ as input. Indeed, this would be very expensive, since the size of the global state space grows exponentially with the number of concurrent processes. It is much cheaper to take advantage of compositionality: by Theorem 2.1, $\pi \in L(A_1||\ldots||A_n) \setminus L(P_1||\ldots||P_n)$ iff, for some i, $\pi\restriction_{Act_i} \in L(A_i) \setminus L(P_i)$. It then suffices to apply abstraction refinement to this particular A_i, since $\pi\restriction_{Act_i} \notin L(A_i')$ implies that $\pi \notin L(A_1||\ldots||A_i'||\ldots||A_n)$. The advantage of this approach follows from the fact that the computational effort required to identify A_i grows only linearly with the number of concurrent components.

5 Experimental Results

Our experiments were carried out with two broad goals in mind. The first goal was to compare the overall effectiveness of the proposed two-level CEGAR approach, particularly insofar as memory usage is concerned. The second goal was to verify the effectiveness of our LTS abstraction scheme by itself. We carried out experiments over 36 examples, of which 26 were sequential programs and 10 were concurrent programs. Each example consisted of an implementation (a C program) and a specification (provided separately as an LTS). All of the experiments were carried out on an AMD Athlon 1800 XP machine with 3 GB RAM running RedHat 7.1.

Example	**LOC**	**Description**	**PredOnly**			**BothAbst**		
			State	*Iter*	*Time*	*State*	*Iter*	Time
lock-y	27	*pthread_mutex_lock* (pthread)	26	1	52	16	3	54
unlock-y	24	*pthread_mutex_unlock* (pthread)	27	1	51	13	2	56
socket-y	60	*socket* (socket)	187	3	1752	44	25	2009
sock_alloc-y	24	*sock_alloc* (socket)	50	2	141	14	4	154
sys_send-y	4	*sys_send* (socket)	7	1	92	6	1	93
sock_sendmsg-y	11	*sock_sendmsg* (socket)	23	1	108	14	3	113
lock-n	27	modified *pthread_mutex_lock*	23	1	59	14	2	61
unlock-n	24	modified *pthread_mutex_unlock*	27	1	61	12	2	66
sock_alloc-n	24	modified *sock_alloc*	47	1	103	9	1	106
sock_sendmsg-n	11	modified *sock_sendmsg*	21	1	96	10	1	97

All times are in milliseconds

Fig. 3. Summary of results for Linux Kernel code. **LOC** and **Description** denote the number of lines of code and a brief description of the benchmark source code. The measurements for *PIter* and *LIter* have been omitted because they are insignificant.

Each example was verified twice, once with only the low-level abstraction, and once with the full two-level algorithm. Tests that used only the low-level predicate abstraction refinement scheme are marked by *PredOnly* in our results tables, whereas tests that also incorporated our LTS action-guided abstraction refinement procedure are marked by *BothAbst*. Both schemes started out with the same initial sets of predicates. For each experiment we measured several quantities: (i) the size of the final state space on which the property was proved/disproved,[3] (ii) the number of predicate refinement iterations required, (iii) the number of LTS refinement iterations required, (iv) the total number of refinement iterations required, and (v) the total time required. In the tables summarizing our results, these measurements are reported in columns named respectively *State, PIter, LIter, Iter* and *Time*.

Unix Kernel examples

The first set of examples were meant to examine how our approach works on a wide spectrum of implementations. We chose ten code fragments from the Linux Kernel 2.4.0. Corresponding to each code fragment we constructed a specification from the Linux man pages. For example, the specification in 'socket-y' states that the socket system call either properly allocates internal data structures for a new socket and returns 1, or fails to do so and returns an appropriate negative error value. The summary of our results on these examples is presented in Figure 3.

OpenSSL Examples

The next set of examples was aimed at verifying larger pieces of code. We designed a set of 26 benchmarks to check various properties of the OpenSSL version 0.9.6c source code, which is a popular open source implementation of the SSL protocol used for secure data transfer over the internet. In particular we used the source code implementing the *handshake* that occurs when an SSL client and server attempt to establish a connection. The source code is accordingly divided into two parts, `SrvrCode` and `ClntCode`, that implement the server and client components respectively. The specifications were derived from the official SSL design documents. For example, the specification for 'ssl-1' states that the handshake is always initiated by the client.

The first 16 examples are sequential implementations, examining different properties of `SrvrCode` and `ClntCode` separately. Each of these examples contains about 350 comment-free LOC. The results for these are summarized in Figure 4. The remaining 10 examples test various properties of `SrvrCode` and `ClntCode` when executed together. These examples are concurrent and consist of about 700 LOC. All OpenSSL benchmarks other than `srvr-7` passed the property. The results are summarized in Figure 5. In terms of state space size, the two-level refinement scheme outperforms the one-level scheme by

[3] Note that, since our abstraction-refinement scheme produces increasingly refined models, and since we reuse memory from one iteration to the next, the size of the final state space represents the *total* memory used.

Example	**PredOnly**					**BothAbst**					**Gain**
	State(S1)	*PIter*	*LIter*	*Iter*	*Time*	*State(S2)*	*PIter*	*LIter*	*Iter*	*Time*	*S1/S2*
srvr-1	563	7	0	7	127	151	7	191	198	142	3.73
srvr-2	323	9	0	9	134	172	9	307	316	156	1.89
srvr-3	362	21	0	21	212	214	20	850	870	263	1.69
srvr-4	227	1	0	1	25	19	1	0	1	23	11.94
srvr-5	3204	98	0	98	1284	878	53	6014	6067	6292	3.65
srvr-6	2614	121	0	121	1418	559	113	9443	9556	6144	4.68
srvr-7	2471	40	0	40	517	662	34	3281	3315	2713	3.73
srvr-8	2614	60	0	60	750	455	37	3158	3195	1992	5.75
clnt-1	402	18	0	18	174	176	19	506	525	209	2.28
clnt-2	408	18	0	18	194	185	16	651	667	217	2.21
clnt-3	633	51	0	51	405	263	58	3078	3136	688	2.41
clnt-4	369	28	0	28	232	193	33	987	1020	306	1.91
clnt-5	318	15	0	15	166	172	13	398	411	182	1.85
clnt-6	323	20	0	20	190	236	21	644	665	242	1.37
clnt-7	323	20	0	20	188	160	20	556	576	221	2.02
clnt-8	314	16	0	16	168	264	16	570	586	215	1.19

All times are in seconds

Fig. 4. Summary of results for sequential OpenSSL examples.

factors ranging from 2 to 136. The savings for the concurrent examples are significantly higher than for the sequential ones. We expect these savings to increase with the number of concurrent components in the implementation.

Although our aim to reduce the size of the state space was achieved, our implementation of the two-level algorithm shows an increase in time over that of the one-level scheme. However, we believe that this situation can be redressed through engineering optimizations of MAGIC. For instance, not only is MAGIC currently based on explicit state enumeration, but also in each iteration it performs the entire verification from scratch. As is evident from our results, the majority of iterations involve LTS refinement. Since the latter only induces a local change in the transition system, the refined model is likely to differ marginally from the previous one. Therefore much of the work done during verification in the previous iteration could be reused. We plan to investigate the possibility of doing *incremental* verification and will report on our findings in the final version of this article.

Example	**PredOnly**					**BothAbst**					**Gain**
	State(S1)	*PIter*	*LIter*	*Iter*	*Time*	*State(S2)*	*PIter*	*LIter*	*Iter*	*Time*	*S1/S2*
ssl-1	108659	8	0	8	243	16960	8	268	276	529	6.41
ssl-2	95535	9	0	9	226	15698	9	331	340	608	6.09
ssl-3	69866	24	0	24	449	23865	19	828	847	1831	2.93
ssl-4	43811	1	0	1	51	323	1	0	1	55	135.64
ssl-5	108659	7	0	7	217	16006	6	186	192	384	6.79
ssl-6	162699	12	0	12	366	18297	9	375	384	792	8.89
ssl-7	167524	23	0	23	599	31250	24	1441	1465	4492	5.36
ssl-8	60602	9	0	9	227	17922	10	434	444	852	3.38
ssl-9	313432	115	0	115	3431	50274	63	3660	3723	15860	6.23
ssl-10	123520	23	0	23	430	23460	21	926	947	2139	5.27

All times are in seconds

Fig. 5. Summary of results for concurrent OpenSSL examples.

6 Conclusions and Future Work

Despite significant research and advancement, automated verification of concurrent programs remains an important, yet elusive, goal. In this paper we presented an approach to automatically and compositionally verify concurrent C programs against safety properties. These concurrent implementations consist of several sequential C programs which communicate via blocking message-passing. Our approach is an instantiation of the CEGAR paradigm, and incorporates two levels of abstraction, which respectively aggregate states according to the values of local variables, and observable events. Experimental results with our tool MAGIC suggest that this scheme effectively combats the state space explosion problem. In all our benchmarks, the two-level algorithm achieved significant reductions in state space (in one case by over two orders of magnitude) compared to the single-level predicate abstraction scheme.

We are currently engaged in extending MAGIC to handle the proprietary implementation of a large industrial controller for a metal casting plant. This code consists of over 30,000 lines of C and incorporates up to 25 concurrent threads which communicate through shared variables. Adapting MAGIC to handle shared memory is therefore one of our priorities. Not only will this enable us to test our tool on the many available shared-memory-based benchmarks, but it will also allow us to compare MAGIC with other similar tools (such as BLAST) which already use shared memory for communication.

Finally, we intend to explore the possibility of adapting our two-level CEGAR scheme to different types of conformance relations such as simulation and bisimulation, so as to handle a wider range of specifications.

References

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

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

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

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

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

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

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

[8] 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.

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

[10] P. Chauhan, E. M. Clarke, J. H. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Proceedings of FMCAD*, pages 33–51, 2002.

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

[12] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons from branching time temporal logic. In *Proceedings of the Workshop on Logics of Programs*, volume 131, pages 52–71. Springer LNCS, 1982.

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of CAV*, volume 1855, pages 154–169. Springer LNCS, 2000.

[14] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *Proceedings of TOPLAS*, pages 1512–1542, September 1994.

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

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

[17] M. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proceedings of CAV*, pages 293–304, 1998.

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

[19] 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*, pages 238–252, 1977.

[20] D. Dams and K. S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *Proceedings of VMCAI*, volume 2575. Springer LNCS, 2003.

[21] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Proceedings of CAV*, volume 1633, pages 160–171. Springer LNCS, 1999.

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

[23] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proceedings of CAV (to appear)*. Springer LNCS, 2003.

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

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

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

[27] R. P. Kurshan. Analysis of discrete event coordination. In *Proceedings REX Workshop 89*, volume 430, pages 414–453. Springer LNCS, 1989.

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

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

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

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

[32] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *Proceedings of CAV*, volume 1855, pages 435–449. Springer LNCS, 2000.

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

[34] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

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

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

[37] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of CAV*, volume 1254, pages 72–83. Springer LNCS, 1997.

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