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

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.

©2003 Published by Elsevier Science B. V.
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-by-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 ac-
cording 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 (Modular Analysis of proGrams In 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 $\langle S, \text{init}, \text{Act}, T \rangle$ with $S$ a finite set of states, $\text{init} \in S$ an initial state, $\text{Act}$ 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 $M = \langle S, \text{init}, \text{Act}, T \rangle$.

A trace $\pi$ 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 $a_1 \ldots a_n \in \text{Act}^*$ such that, for some sequence $s_0 \ldots s_n$ of states of $M$ (with $s_0 = \text{init}$) we have $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$. We refer to the underlying sequence of states $s_0 \ldots s_n$ as the path in $M$ corresponding to the trace $a_1 \ldots a_n$.

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

For a trace $\pi = a_1 \ldots a_n \in \text{Act}^*$ and $s, t \in S$ two states of $M$, we write $s \xrightarrow{\pi} t$ to indicate that $t$ is reachable from $s$ through $\pi$, i.e., that there exist states $s_0 \ldots s_n$ with $s = s_0$ and $t = s_n$, such that $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n$.

Given a state $s \in S$ and a trace $\pi \in \text{Act}^*$, let $\text{Reach}(M, s, \pi) = \{t \in S \mid s \xrightarrow{\pi} t\}$ stand for the set of states reachable from $s$ through $\pi$. We overload this notation by setting, for a set of states $Q \subseteq S$, $\text{Reach}(M, Q, \pi) = \{t \in S \mid \exists q \in Q, q \xrightarrow{\pi} t\}$; this represents the set of states reachable through $\pi$ from some state in $Q$.

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

Let $M_1 = \langle S_1, \text{init}_1, \text{Act}_1, T_1 \rangle$ and $M_2 = \langle S_2, \text{init}_2, \text{Act}_2, T_2 \rangle$ be two LTSs. Their parallel composition $M_1 || M_2 = \langle S_1 \times S_2, (\text{init}_1, \text{init}_2), \text{Act}_1 \cup \text{Act}_2, T_1 || T_2 \rangle$ is defined so that $\langle (s_1, s_2), a, (t_1, t_2) \rangle \in T_1 || T_2$ iff one of the following holds:

(i) $a \in \text{Act}_1 \setminus \text{Act}_2$ and $s_1 \xrightarrow{a} t_1$ and $s_2 = t_2$.
(ii) $a \in \text{Act}_2 \setminus \text{Act}_1$ and $s_2 \xrightarrow{a} t_2$ and $s_1 = t_1$.
(iii) $a \in \text{Act}_1 \cap \text{Act}_2$ 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 $\text{Act}_i = \text{Act}_i'$. If, for each $1 \leq i \leq n$, 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 $\text{Act}_1, \ldots, \text{Act}_n$, and let $\pi$ be any trace. Then $\pi \in L(M_1||\ldots||M_n)$ iff, for each $1 \leq i \leq n$, we have $\pi|_{\text{Act}_i} \in L(M_i)$. 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 $\text{Act}_i$, 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 $\text{Spec}$, where the latter is expressed as an LTS. We use trace containment as our notion of conformance: the concurrent program meets its specification iff $L(C_1||\ldots||C_n) \subseteq L(\text{Spec})$.

3 Two-Level Counterexample-Guided Abstraction Refinement

Consider a concurrent C program $C_1||\ldots||C_n$ and a specification $\text{Spec}$. 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 $C_1, \ldots, C_n$ and specification $Spec$

**Output:** $\langle C_1 \rangle \ldots \langle C_n \rangle$ satisfies $Spec$ or

counterexample $\pi \in L(C_1 \ldots \langle C_n \rangle) \setminus L(Spec)$

**Predicate abst.:** create LTSs $P_1, \ldots, P_n$ with $L(C_i) \subseteq L(P_i)$

**Action-guided abst.:** create LTSs $A_1, \ldots, A_n$ with $L(P_i) \subseteq L(A_i)$

repeat

if $L(A_1 \ldots \langle A_n \rangle) \subseteq L(Spec)$ return $\langle C_1 \rangle \ldots \langle C_n \rangle$ satisfies $Spec$

else

extract counterexample $\pi \in L(A_1 \ldots \langle A_n \rangle) \setminus L(Spec)$

if $\pi \in L(P_1 \ldots \langle P_n \rangle)$

if $\pi \in L(C_1 \ldots \langle C_n \rangle)$ return $\pi$

else

do predicate abstraction refinement of $P_1, \ldots, P_n$

else

adjust or create new abstractions $A_1, \ldots, A_n$

end repeat.

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:

$$L(A_1 \ldots \langle A_n \rangle) \subseteq L(Spec) \Rightarrow L(C_1 \ldots \langle C_n \rangle) \subseteq L(Spec)$$

The converse need not hold: it is possible for a trace $\pi \notin Spec$ to belong to $L(A_1 \ldots \langle A_n \rangle)$ but not to $L(C_1 \ldots \langle C_n \rangle)$. Such a spurious counterexample is then eliminated, either by suitably refining the $A_i$’s (if $\pi \notin L(P_1 \ldots \langle P_n \rangle)$), 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 $L(C_i) \subseteq L(P_i) \subseteq L(A_i)$, where primed terms denote refined processes. Note that, according to Theorem 2.1, we can check whether $\pi \in L(P_1 \ldots \langle P_n \rangle)$ and whether $\pi \in L(C_1 \ldots \langle C_n \rangle)$ 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) $L(P) \subseteq L(A') \subseteq L(A)$,
(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: $t \in [s]_A^0$ iff $enabled(t) = enabled(s)$. We then let $S^0_A = \{[s]_A^0 | s \in S\}$. Again, this uniquely defines our initial abstraction $A^0$, the construction marked $\dagger$ 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 $\bigcup\{[s']_B | s' \in Reach(P, s, a)\} = \bigcup\{[t']_B | t' \in \bigcup\{[s']_B | s' \in Reach(P, t, a)\}\}$. 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 $\pi$. This is achieved by repeatedly splitting states of $A$ along abstract paths which accept $\pi$. The
Input: abstraction $A$ of $P$ (with $L(P) \subseteq L(A)$) and
trace $\pi = a_1 \ldots a_m \in L(A) \setminus L(P)$

Output: refined abstraction $A'$ (with $L(P) \subseteq L(A') \subset L(A)$) and
$\pi \notin L(A')$

while there exists some abstract path $u_0 \xrightarrow{a_1} \ldots \xrightarrow{a_m} u_m$ in $A$ do
  let reachable states = \{init\} /* init = initial state of $P$ */
  let $j = 1$
  while reachable states ≠ 0 do
    let reachable states = Reach($P$, reachable states, $a_j$) ∩ $u_j$
    let $j = j + 1$
  endwhile
  let $A = Abs(Split(S_A, u_{j-2}, a_{j-1})$ /* $S_A$ = 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 $\pi \notin L(P)$, Reach($P$, init, $\pi$) = ∅, 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 reachable states = \{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 $i$, $\pi \upharpoonright_{Act_i} \in L(A_i) \setminus L(P_i)$. It then suffices to apply abstraction refinement to this particular $A_i$, since $\pi \upharpoonright_{Act_i} \notin L(A_i')$ 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.

<table>
<thead>
<tr>
<th>Example</th>
<th>LOC</th>
<th>Description</th>
<th>PredOnly</th>
<th>BothAbst</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>State</td>
<td>Iter</td>
</tr>
<tr>
<td>lock-y</td>
<td>27</td>
<td>$pthread_mutex_lock$ (pthread)</td>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>unlock-y</td>
<td>24</td>
<td>$pthread_mutex_unlock$ (pthread)</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>socket-y</td>
<td>60</td>
<td>$socket$ (socket)</td>
<td>187</td>
<td>3</td>
</tr>
<tr>
<td>sock alloc-y</td>
<td>24</td>
<td>$sock_alloc$ (socket)</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>sys send-y</td>
<td>4</td>
<td>$sys_send$ (socket)</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>sock sendmsg-y</td>
<td>11</td>
<td>$sock_send_msg$ (socket)</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>lock-n</td>
<td>27</td>
<td>modified $pthread_mutex_lock$</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>unlock-n</td>
<td>24</td>
<td>modified $pthread_mutex_unlock$</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>sock alloc-n</td>
<td>24</td>
<td>modified $sock_alloc$</td>
<td>47</td>
<td>1</td>
</tr>
<tr>
<td>sock sendmsg-n</td>
<td>11</td>
<td>modified $sock_send_msg$</td>
<td>21</td>
<td>1</td>
</tr>
</tbody>
</table>

*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, *SrvcCode* and *ClnetCode*, 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 *SrvcCode* and *ClnetCode* 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 *SrvcCode* and *ClnetCode* when executed together. These examples are concurrent and consist of about 700 LOC. All OpenSSL benchmarks other than *srvc-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.
<table>
<thead>
<tr>
<th>Example</th>
<th>PredOnly</th>
<th>BothAlst</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>State(S1) FIter LIter Iter Time</td>
<td>State(S2) FIter LIter Iter Time</td>
<td>S1/S2</td>
</tr>
<tr>
<td>srrv-1</td>
<td>563 7 0 7 127</td>
<td>151 7 191 198 142</td>
<td>3.73</td>
</tr>
<tr>
<td>srrv-2</td>
<td>323 9 0 9 134</td>
<td>172 9 307 316 156</td>
<td>1.89</td>
</tr>
<tr>
<td>srrv-3</td>
<td>362 21 0 21 212</td>
<td>214 20 850 870 263</td>
<td>1.69</td>
</tr>
<tr>
<td>srrv-4</td>
<td>227 1 0 1 25</td>
<td>19 1 0 1 23</td>
<td>11.94</td>
</tr>
<tr>
<td>srrv-5</td>
<td>3204 98 0 98 1284</td>
<td>878 53 6014 6067 6292</td>
<td>3.65</td>
</tr>
<tr>
<td>srrv-6</td>
<td>2614 121 0 121 1418</td>
<td>559 113 9443 9556 6144</td>
<td>4.68</td>
</tr>
<tr>
<td>srrv-7</td>
<td>2471 40 0 40 517</td>
<td>662 34 3281 3315 2713</td>
<td>3.73</td>
</tr>
<tr>
<td>srrv-8</td>
<td>2614 60 0 60 750</td>
<td>455 37 3158 3195 1992</td>
<td>5.75</td>
</tr>
<tr>
<td>chnt-1</td>
<td>402 18 0 18 174</td>
<td>176 19 506 525 209</td>
<td>2.28</td>
</tr>
<tr>
<td>chnt-2</td>
<td>408 18 0 18 194</td>
<td>185 16 651 667 217</td>
<td>2.21</td>
</tr>
<tr>
<td>chnt-3</td>
<td>633 51 0 51 405</td>
<td>263 58 3078 3136 688</td>
<td>2.41</td>
</tr>
<tr>
<td>chnt-4</td>
<td>369 28 0 28 232</td>
<td>193 33 987 1020 306</td>
<td>1.91</td>
</tr>
<tr>
<td>chnt-5</td>
<td>318 15 0 15 166</td>
<td>172 13 398 411 182</td>
<td>1.85</td>
</tr>
<tr>
<td>chnt-6</td>
<td>323 20 0 20 190</td>
<td>236 21 644 665 242</td>
<td>1.37</td>
</tr>
<tr>
<td>chnt-7</td>
<td>323 20 0 20 188</td>
<td>160 20 556 576 221</td>
<td>2.02</td>
</tr>
<tr>
<td>chnt-8</td>
<td>314 16 0 16 168</td>
<td>264 16 570 586 215</td>
<td>1.19</td>
</tr>
</tbody>
</table>

*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.
<table>
<thead>
<tr>
<th>Example</th>
<th>PredOnly</th>
<th>BothAbst</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>State(S1)</td>
<td>PIter</td>
<td>LIter</td>
</tr>
<tr>
<td>ssi-1</td>
<td>108659</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>ssi-2</td>
<td>95535</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>ssi-3</td>
<td>69866</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>ssi-4</td>
<td>43811</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ssi-5</td>
<td>108659</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>ssi-6</td>
<td>162699</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>ssi-7</td>
<td>167524</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>ssi-8</td>
<td>60602</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>ssi-9</td>
<td>313432</td>
<td>115</td>
<td>0</td>
</tr>
<tr>
<td>ssi-10</td>
<td>123520</td>
<td>23</td>
<td>0</td>
</tr>
</tbody>
</table>

*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


