Research Institute for Advanced Computer Science

Technical Report

Abstraction and Assume-Guarantee Reasoning for Automated Software Verification

Sagar Chaki

Edmund Clarke

Dimitra Giannakopoulou

Corina S. Păsăreanu

Technical Report 05.02

October 2004

Abstraction and Assume-Guarantee Reasoning for Automated Software Verification

S. Chaki, *Carnegie Mellon SEI*
E. Clarke, *Carnegie Mellon University*
D. Giannakopoulou, *RIACS / USRA*
C.S. Păsăreanu, *Kestrel Technology LLC*

Technical Report 05.02

October 2004

Compositional verification and abstraction are the key techniques to address the state explosion problem associated with model checking of concurrent software. A promising compositional approach is to prove properties of a system by checking properties of its components in an assume-guarantee style. This article proposes a framework for performing abstraction and assume-guarantee reasoning of concurrent C code in an incremental and fully automated fashion. The framework uses predicate abstraction to extract and refine finite state models of software and it uses an automata learning algorithm to incrementally construct assumptions for the compositional verification of the abstract models. The framework can be instantiated with different assume-guarantee rules. We have implemented our approach in the COMFORT reasoning framework and we show how COMFORT out-performs several previous software model checking approaches when checking safety properties of non-trivial concurrent programs.

This work was supported in part by the National Aeronautics and Space Administration under Cooperative Agreement NCC 2-1426 with the Universities Space Research Association (USRA).

Abstraction and Assume-guarantee Reasoning for Automated Software Verification

S. Chaki[1], E. Clarke[2], D. Giannakopoulou[3], and C.S. Păsăreanu[4]

[1] Carnegie Mellon Software Engineering Institute
[2] Carnegie Mellon University
[3] RIACS, NASA Ames Research Center, Moffett Field, CA, USA
[4] Kestrel Technology LLC, NASA Ames Research Center, Moffett Field, CA, USA

Abstract. Compositional verification and abstraction are the key techniques to address the state explosion problem associated with model checking of concurrent software. A promising compositional approach is to prove properties of a system by checking properties of its components in an assume-guarantee style. This article proposes a framework for performing abstraction and assume-guarantee reasoning of concurrent C code in an incremental and fully automated fashion. The framework uses predicate abstraction to extract and refine finite state models of software and it uses an automata learning algorithm to incrementally construct assumptions for the compositional verification of the abstract models. The framework can be instantiated with different assume-guarantee rules. We have implemented our approach in the ComFoRT reasoning framework and we show how ComFoRT out-performs several previous software model checking approaches when checking safety properties of non-trivial concurrent programs.

1 Introduction

The verification of concurrent software is acknowledged as an important and difficult problem. Model checking [12] is becoming a popular technique for software verification [3, 4, 18, 24]. Given some formal description of a system and of a required property, model checking automatically determines whether the property is satisfied by the system. The limitation of the approach is the *state space explosion* problem, where the number of reachable states of a concurrent system increases exponentially with the number of its components.

It is generally recognized that compositional reasoning and abstraction are the key techniques to combat state-space explosion. Together these two paradigms enable us to reduce the problem of verifying a large system into several smaller tasks of verifying simpler systems [28]. We present a framework that uses assume-guarantee style compositional reasoning and predicate abstraction for the automated verification of concurrent software. Our work is done in the context of concurrent C programs with synchronous (blocking) message passing communication, e.g. client server protocols, schedulers, telecommunication applications, NASA autonomy software, etc. We consider safety properties that describe the legal (and illegal) sequences of actions that a system is allowed to perform.

Assume-guarantee reasoning checks properties of a system by proving properties of its components under assumptions that these components make about their respective environments [27, 30]. Intuitively, an assumption characterizes all contexts in which a component is expected to operate correctly. These assumptions also need to be discharged, by verifying that the rest of the system indeed satisfies them. This style of reasoning is often non-trivial, typically requiring human input to determine appropriate assumptions. Therefore, to apply assume-guarantee techniques on software of industrial complexity it is imperative that we construct appropriate assumptions in a completely automated manner.

The proposed framework builds on our previous work which uses learning to automate assume-guarantee style verification [14]. That work was done in the context of checking two components expressed as *finite state* labeled transition systems. In contrast, this article presents an *implementation* and *evaluation* of automated assume-guarantee verification for *infinite state* systems (using abstraction and refinement). Moreover, we address the verification of programs written in an advanced programming language (i.e. C), in the context of two or more components using symmetric and non-symmetric assume-guarantee rules.

To check that a system made up of several software components satisfies a property, our framework follows the iterative counterexample guided abstraction refinement (CEGAR) paradigm [11]. In each iteration, we first use *predicate abstraction* [19] to build automatically finite state models which are conservative abstractions (with respect to safety properties) of the software components. We then use the L^* *learning algorithm* [2, 31] to build incrementally assumptions that are used for the compositional verification of the abstract models.

If the outcome of the verification is that the property holds on the abstract model, we conclude that the property also holds on the original system. Otherwise, we analyze the returned counterexample to see if it corresponds to a real error or if it is a spurious behavior introduced by the abstraction process. In the latter case, we use the counterexample to *refine* automatically the appropriate abstract models and repeat the CEGAR loop. The process is carried out component wise, without it ever being necessary to build the state space of the whole system. Furthermore, our verification procedure is flexible and can be instantiated with different assume-guarantee rules.

There has been substantial work on compositional reasoning [1, 13, 21, 23] and abstraction [3, 15] for software verification. However, there are few instances where these two techniques have been combined effectively to enable automated analysis of complex concurrent software. The MAGIC tool [7, 10] uses a two-level abstraction scheme for the compositional verification of concurrent, message-passing C programs. We show (in Section 6) how our framework outperforms MAGIC when checking several properties in a non-trivial concurrent C program (74,000 LOC). The BLAST tool [6] uses predicate abstraction and assume-guarantee reasoning for checking *race conditions* in multi-threaded C code. In contrast to our work, it targets *shared memory* communicating programs, and therefore it uses a different style of assume guarantee rule. Moreover, the approach used by BLAST is not based on learning.

We summarize our contributions (and the organization of this article) as follows. We describe a novel framework (Section 3) that uses assume-guarantee reasoning (Section 4), predicate abstraction and automated CEGAR (Section 5) for the compositional analysis of concurrent, message passing C programs. We present instantiations of the framework with several assume-guarantee rules. Other rules could be used, provided that they are sound in the context of our framework. We provide an implementation and case studies showing the merits of our approach (Section 6); we also evaluate the scalability of our approach with increasing number of components by comparing our implementation with the state of the art model checker SPIN [32].

We end the paper with an overview of related work (Section 7) and conclusions (Section 8). In the next section we provide some background information.

2 Background

We use Labeled Transition Systems (LTSs) to model the behavior of communicating components in an (abstracted) concurrent system. As described in Section 4, we use proof rules that require the "complement" of an LTS. LTSs are not closed under complementation (their languages are prefix-closed), so we need to define here a more general class of finite state machines (FSMs).

Let $\mathcal{A}ct$ be the universal set of observable actions and let τ denote a local action *unobservable* to a component's environment.

Definition 1 (FSM). *An FSM M is a tuple $\langle Q, \alpha M, \delta, q0, F\rangle$ where: (i) Q is a non-empty finite set of states, (ii) $\alpha M \subseteq \mathcal{A}ct$ is a finite set of observable actions called the* alphabet *of M, (iii) $\delta \subseteq Q \times (\alpha M \cup \{\tau\}) \times Q$ is a transition relation, (iv) $q0 \in Q$ is the initial state, and (v) $F \subseteq Q$ is a set of accepting states.*

An LTS is a special instance of an FSM for which all states are accepting. An FSM $M = \langle Q, \alpha M, \delta, q0, F\rangle$ is *non-deterministic* if it contains τ-transitions or if $\exists (q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, M is *deterministic.*

Traces A word (or trace) is an element of $\mathcal{A}ct^*$. For an FSM M and a word t, we use $\hat{\delta}(q, t)$ to denote the set of states that M can reach after reading t starting at state q. A word t is said to be *accepted* by an FSM $M = \langle Q, \alpha M, \delta, q0, F\rangle$ if $\hat{\delta}(q0, t) \cap F \neq \emptyset$. The *language accepted by M*, denoted $\mathcal{L}(M)$ is the set $\{t \mid \hat{\delta}(q0, t) \cap F \neq \emptyset\}$. For $\Sigma \subseteq \mathcal{A}ct$, we use $t{\upharpoonright}\Sigma$ to denote the trace obtained by removing from t all occurrences of actions $a \notin \Sigma$. We use $[t, \Sigma]$ to denote the FSM whose alphabet is Σ and whose language is the singleton set $\{t\}$.

Definition 2 (Parallel Composition). *Let $M_1 = \langle Q_1, \alpha M_1, \delta_1, q0_1, F_1\rangle$ and $M_2 = \langle Q_2, \alpha M_2, \delta_2, q0_2, F_2\rangle$ be two FSMs. Then $M_1 \parallel M_2$ is an FSM $M = \langle Q, \alpha M, \delta, q0, F\rangle$, where: (i) $Q = Q_1 \times Q_2$, (ii) $\alpha M = \alpha M_1 \cup \alpha M_2$, (iii) $F = F_1 \times F_2$, (iv) $q0 = (q0_1, q0_2)$, and (v) $((s_1, s_2), a, (s_1', s_2')) \in \delta$ iff $((s_1, a, s_1') \in \delta_1 \wedge s_2 = s_2' \wedge a \notin \alpha M_2) \vee (s_2, a, s_2') \in \delta_2 \wedge s_1 = s_1' \wedge a \notin \alpha M_1) \vee ((s_1, a, s_1') \in \delta_1 \wedge (s_2, a, s_2') \in \delta_2 \wedge a \neq \tau)$.*

The parallel composition operator $\|$ is a commutative and associative operator that combines the behavior of two components by synchronizing the actions common to their alphabets and interleaving the remaining actions.

Properties and Satisfiability A property is defined as an LTS P, whose language $\mathcal{L}(P)$ defines the set of acceptable behaviors over αP. An FSM M satisfies P, denoted as $M \models P$, if and only if $\forall t \in \mathcal{L}(M) \centerdot t{\restriction}\alpha P \in \mathcal{L}(P)$.

Completion An FSM is complete with respect to some alphabet if every state has an outgoing transition for each action in the alphabet. Completion typically introduces a new non-accepting state and it adds transitions to the new state so that the automaton becomes completes.

Complementation The complement of an FSM (or an LTS) M, denoted $\overline{M}$, is an FSM that accepts the complement of M's language. It is constructed by first making M deterministic, subsequently completing it with respect to αM, and finally turning all accepting states into non-accepting ones, and vice-versa.

Weakest Environment Assumption We will use the notion of the *weakest environment assumption* [17]. For any FSM M and property P, let $WA(M, P)$ denote the weakest environment under which M can achieve P. In other words, $WA(M, P)$ is an FSM such that, for any FSM A, $M\|A \models P$ iff $A \models WA(M, P)$.

The L* Algorithm The learning algorithm (L*) used by our approach was developed by Angluin [2] and later improved by Rivest and Schapire [31]. L* learns an unknown regular language U over an alphabet Σ and produces a deterministic FSM C such that $\mathcal{L}(C) = U$. L* works by incrementally producing a sequence of candidate deterministic FSMs C_1, C_2, ... converging to C. In order to learn U, L* needs a *Teacher* to answer two types of questions. The first type is a *membership query*, consisting of a string $\sigma \in \Sigma^*$; the answer is *true* if $\sigma \in U$, and *false* otherwise. The second type of question is a *conjecture*, i.e. a candidate deterministic FSM C_i whose language the algorithm believes to be identical to U. The answer is *true* if $\mathcal{L}(C_i) = U$. Otherwise the Teacher returns a counterexample, i.e. a string σ in the symmetric difference of $\mathcal{L}(C_i)$ and U.

At a higher level, L* creates a table where it incrementally records whether strings in Σ^* belong to U. It does this by making membership queries to the Teacher. At various stages L* decides to make a conjecture. It constructs a deterministic FSM C_i based on the information contained in the table and asks the Teacher whether the conjecture C_i is correct. If it is, the algorithm terminates. Otherwise, L* uses the counterexample returned by the Teacher to extend the table with strings that witness differences between $\mathcal{L}(C_i)$ and U.

L* is guaranteed to terminate with a minimal automaton C for the unknown language U. Moreover, each candidate FSM C_i that L* constructs is smallest, in the sense that any other deterministic FSM consistent with the table from which C_i was constructed has at least as many states as C_i. The candidates conjectured by L* strictly increase in size; each candidate is smaller than the next one, and all incorrect candidates are smaller than C. Therefore, if C has n states, L* makes at most $n - 1$ incorrect conjectures.

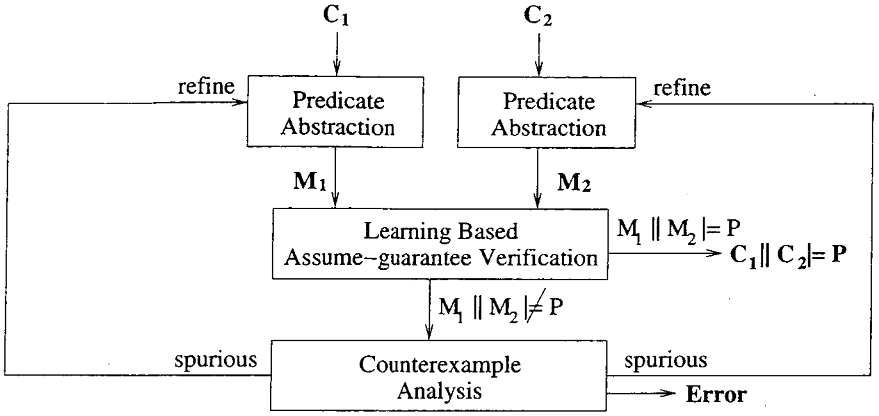

Fig. 1. Assume-guarantee verification of source code

3 Framework for Compositional Verification of Software

Our approach advocates a combination of learning based assume-guarantee style reasoning (described in Section 4) and predicate abstraction techniques (described in Section 5) to verify real-life programs.

The approach is illustrated in Figure 1. Consider two message-passing C programs C_1 and C_2, and let P be a safety property describing the legal sequences of actions in $C_1||C_2$ (generalization to more than two components is done as described in Section 4). We want to check $C_1||C_2 \models P$. The state space of each of the two C programs may be very large, or even infinite. Predicate abstraction is used to extract finite conservative models M_1 and M_2 of C_1 and C_2 respectively.

Since the state space of $M_1||M_2$ could still be prohibitively large, we break up the verification of $M_1||M_2$ by using learning based assume-guarantee reasoning. The learning box, which can be instantiated with different assume-guarantee rules, computes in an iterative manner, appropriate assumptions that are necessary for the compositional verification of $M_1||M_2$. Since M_1 and M_2 are finite, the process is guaranteed to terminate, stating either that the property holds for $M_1||M_2$, or returning a counterexample if the property is violated.

If $M_1||M_2 \models P$, we conclude that it is also the case that $C_1||C_2 \models P$, since M_1 and M_2 are conservative abstractions. Otherwise, the counterexample is analyzed to see if it corresponds to an error in $C_1||C_2$, or if it is spurious as a result of the abstraction. In the latter case, the counterexample is exploited to automatically refine the set of predicates used by the predicate abstraction and the process is repeated.

Our framework builds increasingly precise abstract models and assumptions. If time or memory is not sufficient to reach termination, intermediate results may still contain useful information and may be further analyzed. The generated assumptions may be useful in approximating the requirements that a component places on its environment to satisfy certain properties, while the abstract models approximate the behavior of the software components.

4 Assume-guarantee Verification for Finite State Models

In this section we discuss the use of different proof rules in the context of learning based assume-guarantee verification of finite state systems.

4.1 Automated assume-guarantee verification for two components

Rule 1 In our previous work on assumption generation and learning [14], we used the following basic assume-guarantee rule for establishing that a property P holds for the parallel composition of two *finite state* models of software components M_1 and M_2.

$$\begin{array}{l} 1: M_1 \parallel A_{M_1} \models P \\ 2: M_2 \models A_{M_1} \\ \hline \quad M_1 \parallel M_2 \models P \end{array}$$

A_{M_1} denotes an assumption about the environment in which M_1 is placed. The alphabet of A_{M_1} is $(\alpha M_1 \cup \alpha P) \cap \alpha M_2$.

The approach presented in [14] iterates a process based on gradually *learning* an assumption that is strong enough for M_1 to satisfy P but weak enough to be an abstraction of M_2's behavior. The learning process generates candidate assumptions based on queries to component M_1 and on counterexamples obtained by model checking the two premises of the rule, alternately. For finite state systems, this process is guaranteed to terminate stating that the property holds in $M_1 || M_2$ or returning a counterexample that exhibits a property violation.

Rule 2 Rule 1 is not symmetric in its use of the two components. Symmetric rules are interesting in the context of our framework, as their use is expected to lead to earlier termination of the iterative process and to smaller assumptions. Several such rules are presented in [5]. We have implemented the following rule from [5] (other rules could also be easily incorporated).

$$\begin{array}{l} 1: M_1 \parallel A_{M_1} \models P \\ 2: M_2 \parallel A_{M_2} \models P \\ 3: \overline{A_{M_1}} \parallel \overline{A_{M_2}} \models P \\ \hline \quad M_1 \parallel M_2 \models P \end{array}$$

We require $\alpha P \subseteq \alpha M_1 \cup \alpha M_2$ and $\alpha A_{M_1} = \alpha A_{M_2} = (\alpha M_1 \cap \alpha M_2) \cup \alpha P$ (denoted αA). Intuitively, premise 3 ensures that the possible common traces of M_1 and M_2, which are ruled out by the two assumptions, satisfy the property.

Proposition 1. *Rule 2 is sound*[5] *[5].*

The use of Rule 2 in the context of automated learning based assume-guarantee verification is illustrated in Figure 2. L* is used to generate incrementally an assumption for each component such that premises 1 and 2 of Rule 2 hold. Specifically, L* is used to iteratively learn the traces of $WA(M_1, P)$, and $WA(M_2, P)$

[5] Rule 2 is also complete [5].

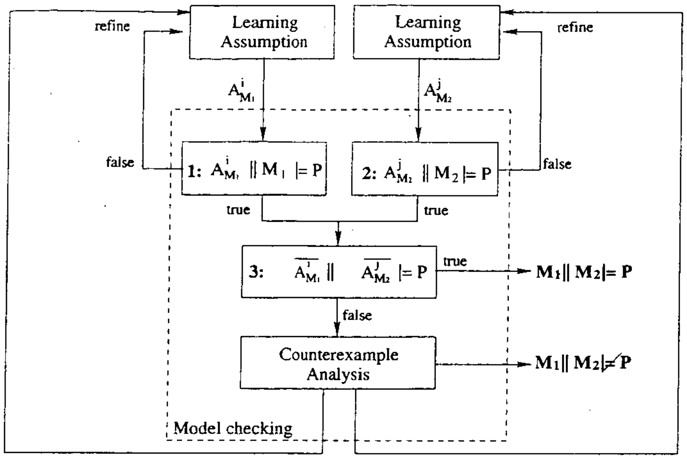

Fig. 2. Learning based assume-guarantee verification

respectively. Conjectures are intermediate assumptions. As in [14], model checking is used to implement the Teacher needed by L*.

At each iteration, L* is used to build approximate assumptions $A^i_{M_1}$ and $A^j_{M_2}$, based on *querying* the system and on the results of the previous iteration. Premise 1 is then checked to determine whether $M_1 || A^i_{M_1} \models P$. If the result is false, then the returned counterexample is not in the language of $WA(M_1, P)$. The counterexample is therefore used to *refine* $A^i_{M_1}$, through learning. Premise 2 is checked similarly, to obtain an assumption $A^j_{M_2}$ such that $M_2 || A^j_{M_2} \models P$.

When the first two premises hold, premise 3 is checked to discharge the assumptions. If premise 3 holds, then, according to the compositional rule, $M_1 \parallel M_2 \models P$. Otherwise, the obtained counterexample t is analyzed as follows.

1. Use model checking to determine if $M_1 || [t, \alpha A] \models P$. If true, then $t \in \mathcal{L}(WA(M_1, P))$, so assumption $A^i_{M_1}$ needs to be *refined* (through learning) and the whole verification process is repeated with a new iteration. Otherwise, get a new counterexample t_1 and go to step 2.
2. Similarly, check $M_2 || [t, \alpha A] \models P$. If true, assumption $A^j_{M_2}$ is *refined* and the whole verification process is repeated with a new iteration. Otherwise, get a new counterexample t_2 and go to step 3.
3. We report any trace from $\hat{t} \in \mathcal{L}([t_1, \alpha A \cup \alpha M_1] || [t_2, \alpha A \cup \alpha M_2])$ as a counterexample to $M_1 || M_2 \models P$.

Proposition 2. *$\hat{t}$ is a counterexample for $M_1 || M_2 \not\models P$.*

4.2 Generalization to n components

So far, we have discussed assume-guarantee reasoning in the context of two components. Assume now that a system consists of n components $M_1 \parallel \cdots \parallel M_n$. We are interested in generalizing the two rules presented above to reason about n components. We have implemented the following approaches.

Generalization for Rule 1 We break the system $M_1 \parallel \cdots \parallel M_n$ into two parts M_1 and $M_2' = M_2 \parallel \cdots \parallel M_n$, and we use Rule 1 for checking (using learning based assume-guarantee reasoning) $M_1 \parallel M_2' \models P$. Then, the second premise of Rule 1 involves checking: $M_2 \parallel \cdots \parallel M_n \models A_{M_1}$, where A_{M_1} is a generated assumption for M_1. To discharge this obligation, our implementation invokes itself recursively, to avoid the state-space explosion that can arise due to the composition of $M_2 \parallel \cdots \parallel M_n$. Note that this enables us to verify $M_1 \parallel ... \parallel M_n \models P$ without ever computing the composition of two or more components. We should note that this generalization approach is mentioned briefly in [14] as a future research direction. We provide here an implementation and an evaluation of its scalability with increasing number of components(cf. Section 6).

Generalization for Rule 2 We generalize Rule 2 as follows.

$$\frac{\begin{array}{r l} 1..n: & M_i \parallel A_{M_i} \models P \\ n+1: & \overline{A_{M_1}} \parallel ... \parallel \overline{A_{M_n}} \models P \end{array}}{M_1 \parallel ... \parallel M_n \models P}$$

This rule is incorporated in a straightforward way into the assume-guarantee framework; the idea is to use learning for each assumption that appears in premises $1..n$ of the rule, and to use the last premise to discharge these assumptions. The drawback of using this rule is that it needs to compute the product of the complements of all the n assumptions to discharge the final premise. One way around this problem is to weaken the final premise. However this could result in a loss of completeness. In other words, a reported counterexample for $M_1||M_2 \not\models P$ might not correspond to a real counterexample.

4.3 Discussion

For finite state systems, the iterative learning process is guaranteed to terminate. This follows from the correctness of L*, which guarantees that if it keeps receiving counterexamples, it will eventually, produce $WA(M_1, P)$ and $WA(M_2, P)$ respectively. Note that the process may terminate before the weakest assumptions are constructed. It terminates as soon as two assumptions have been constructed that are strong enough to discharge the first two premises but weak enough for the third premise to produce conclusive results, i.e. to prove the property or produce a counterexample.

The L* algorithm guarantees that the generated assumptions are minimal; they strictly increase in size, and grow no larger than the weakest assumptions $WA(M_1, P)$ and $WA(M_2, P)$. We should note that, in the worst case, the cost of building directly the weakest assumption for a component is exponential in the size of that component [17]. However, for well designed software, the interfaces between components are usually small. Therefore, assumptions are expected to be *much smaller* than the components that they restrict in the compositional rules. It is in cases like this that we expect our learning based approach to work best. L* approximates the assumptions starting from very small state machines and it runs in time that is polynomial in the size of the automaton it is learning [31]. Therefore, the cost of learning based assume-guarantee verification is expected to be small, as compared to non-compositional model checking or other compositional abstraction techniques (see also the results from Section 6).

4.4 Extensions

Our framework is flexible and it can be extended with other assume-guarantee rules. One candidate rule appears in [29]. The rule is circular and it uses induction over time to break the mutual dependence between components. Its incorporation in our framework is straightforward (similar to Rule 2).

We are well aware that there is no single rule that can be effective in addressing the state space explosion problem in all cases. Therefore, we plan to equip our framework with a library of different assume-guarantee rules and to evaluate their effectiveness on extensive case studies. The framework can use rules that are sound in the context of FSMs with blocking communication. Note that completeness of the rules is not required, although it is desirable. The use of incomplete rules guarantees positive results, but it might yield false counterexamples that can not be used in abstraction-refinement the positive results. To accommodate incomplete rules, further counterexample analysis is needed to determine if the returned counterexample indicates a real error.

Another promising direction is the dynamic use of previously generated assumptions to verify evolving software in the presence of component up-dates or substitutions (see [9]).

5 Abstraction and Refinement for C

Given a (concurrent) C program made up of a number of components $C_1...C_n$, a set of predicates on program variables, and a safety property P, we use predicate abstraction to automatically build finite state LTSs $M_1...M_n$ respectively. These abstract models are *conservative*, i.e. $\mathcal{L}(C_i) \subseteq \mathcal{L}(M_i)$ for $i = 1 \ldots n$. We then check $M_1||...||M_n \models P$. The results obtained from this verification are used to refine the abstract models, if necessary. The abstraction, counterexample validation, and model refinement steps are all performed component-wise. We present an overview of abstraction and refinement here (for a detailed description see [8]). We improve upon [8], by using learning based assume-guarantee reasoning for checking $M_1||...||M_n \models P$ (cf. Section 3).

Model construction Finite state LTSs are computed from the control flow graph of a program in combination with predicate abstraction. Given a set of predicates defined over the state variables of a program, predicate abstraction computes an abstract model that describes the behaviors of the original program in terms of these predicates. To decide properties such as equivalence of predicates, we use theorem provers. Once a finite set of predicates is chosen, the states of the corresponding abstraction are simply valuations of the predicates.

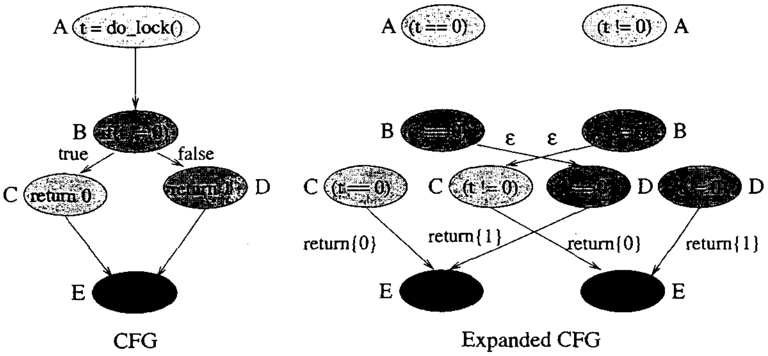

Fig. 3. Predicate abstraction for source code

The transition relation of the abstract system is defined existentially, i.e., we postulate a transition from abstract state A to abstract state B if there are concrete states a and b, associated with A and B respectively, that have a transition from a to b. Figure 3 provides a simple example illustrating these ideas. The left-hand side shows the control flow graph (CFG) of a simple C program with two integer variables x and t. Suppose now that we define a single predicate $P := t = 0$. Then corresponding to each control location we have two abstract states: P can either be *True* or *False*. The right-hand side shows the model that we obtain via predicate abstraction. Transitions are labeled with actions that can represent synchronization messages (absent here), the return values of procedure calls, and internal actions.

The initial set of predicates can be obtained in many ways, the most common being to collect formulas appearing in conditional expressions as well as in the property to be checked. The user is also able to specify predicates of interest, perhaps based on some deeper understanding of the system. New predicates are generated, if needed, in the model refinement phase, described next.

Model refinement The model constructed by predicate abstraction is guaranteed to be a conservative abstraction of the original system, meaning that each behavior in the original system is represented by some behavior in the model, although the model may contain more behaviors. As a result, if the model satisfies a safety property, then so does the original system [12]. However, a counterexample obtained by verifying the model may be spurious. The counterexample is analyzed, and, if it is spurious, it is used to derive additional predicates and construct a new, finer abstraction of the system. The verification is then repeated anew with the refined model. This abstract-verify-analyze counterexample-refine loop (CEGAR) continues until a real counterexample is obtained or the system is verified to be correct. In theory, the CEGAR loop is not guaranteed to terminate. In practice however, it has proved to be quite effective.

One-Level		Two-Level		A-G Rule 1		A-G Rule 2	
Time	Memory	Time	Memory	Time	Memory	Time	Memory
859	841	1259	127	**288**	70	470	**62**
930	1081	2539	165	**448**	76	879	**66**
*	1888	*	212	3262	130	**2840**	**81**
1705	1313	417	62	**276**	65	446	**58**
*	1449	*	226	938	74	**634**	**61**
*	2146	2123	135	759	94	**738**	**71**
1579	1157	674	104	391	79	**290**	**66**
2772	2143	1178	84	406	80	**382**	**71**

Fig. 4. Comparison between A-G based and two-level schemes. Time is in seconds, memory in MB. A * indicates timeout after 1 hr. Best figures are highlighted.

6 Experimental Results

We implemented our algorithms in the ComFoRT reasoning framework [26]. ComFoRT includes, among others, a model checking engine based on the MAGIC tool [7], and hence inherits all the capabilities of MAGIC. In particular ComFoRT can extract finite LTS models from C programs using predicate abstraction and perform counterexample validation and abstraction refinement automatically and modularly as described in Section 5.

In this section, we describe the evaluation of our techniques in the context of two sets of experiments. All our experiments were carried out on an AMD 1800+ XP machine with 3 GB of RAM.

Comparison with MAGIC In the first set of experiments we compared our approach with the one-level and two-level abstraction refinement schemes [10] which were already implemented in MAGIC. Given a concurrent system $C_1||...||C_n$ and a safety property P, the one-level abstraction scheme uses predicate abstraction and refinement to compute (component-wise) corresponding abstract models $M_1...M_n$ and it checks (non-compositionally) if $M_1||...||M_n$ satisfies P. The two-level abstraction scheme combines predicate abstraction (used to compute abstract models $M_1...M_n$) with a second abstraction operating on events, which lumps together the states of the abstract models, if they perform the same set of actions, and refines these partitions according to reachable successor states. This second abstraction is also applied component-wise and it yields successive models that converge to the bisimulation quotient of the original abstract models.

As the source code for our benchmarks we used OpenSSL version 0.9.6c which has about 74,000 LOC. We checked various safety properties of the initial handshake protocol between an SSL server and an SSL client.

Figure 4 summarizes our results. We imposed a time limit of 1 hour and a memory limit of 2.5 GB. As expected the one-level approach shows the worst performance since it leverages the least amount of compositionality. More importantly, we see that the assume-guarantee (A-G) based approaches are also

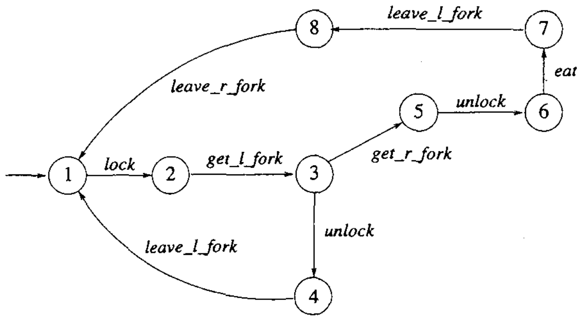

Fig. 5. State machine showing the behavior of each philosopher

consistently better than the two-level approach in terms of both time and memory consumption. In some instances, the A-G based schemes are able to terminate successfully while the two-level approach times out.

Between the two A-G based schemes, the use of Rule 1 enables faster verification but consumes more memory. Intuitively this is because Rule 1 involves composing the concrete system M_2 with the generated assumption (cf. Section 4) while Rule 2 only requires the composition of the (negation of the) two assumptions which are much smaller.

Comparison with SPIN The second set of experiments were aimed at gauging the scalability of our approach with increasing number of components and comparing our implementation with the SPIN [32] model checker. To this end we designed a benchmark based on the classical dining philosophers problem. Each instance of our problem consisted of n philosophers, n forks and a single *lock*.

The behavior of each philosopher is shown in Figure 5. Intuitively, each philosopher behaves as follows. It first attempts to acquire the lock. On success it attempts to grab its left fork. On further success it non-deterministically chooses to either (case 1) go for its right fork or (case 2) release the lock and the left fork and go back to its initial state. In case 1, if it succeeds in getting the right fork, it releases the lock, eats, releases the right fork, then the left fork and goes back to its initial state.

The safety property we attempted to verify states that it should never be the case that all n philosophers simultaneously go for case 1 (i.e., to state 5 in Figure 5) since this would lead to an immediate deadlock. Clearly this is impossible since the lock ensures mutual exclusion. We encoded this system for increasing values of n using both C and Promela (the input language of SPIN). We then verified the safety property using ComFoRT and SPIN. We again imposed a time limit of 1 hour and a memory limit of 2.5 GB.

For SPIN we used the compile time options -DSAFETY and -DMA=444 which lead to reduced memory consumption. Without these options, the performance of SPIN is even worse. For ComFoRT we report results for the non-symmetric Rule 1, generalized as described in Section 4. The use of Rule 2 scales poorly with

n	SPIN				A-G Rule 1			
	Time	*T-Inc*	Memory	*M-Inc*	Time	*T-Inc*	Memory	*M-Inc*
2	4	-	281	-	**2**	-	**6.3**	-
3	**5**	1.25	281	1.00	6	3.00	**6.7**	1.06
4	**7**	1.40	282	1.00	53	8.83	**9.6**	1.43
5	**9**	1.29	282	1.00	80	1.51	**10.5**	1.09
6	**18**	2.00	283	1.00	101	1.26	**11.5**	1.10
7	**53**	2.94	284	1.00	178	1.76	**13.8**	1.20
8	**201**	3.79	287	1.01	235	1.32	**15.5**	1.12
9	846	4.21	299	1.04	**384**	1.63	**19.6**	1.26
10	*	-	339	1.13	**577**	1.50	**23.3**	1.19
11	*	-	529	1.56	**876**	1.52	**28.0**	1.20
12	*	-	800	-	**1307**	1.49	**34.9**	1.25

Fig. 6. Comparison between A-G based scheme and SPIN. Time is in seconds, memory in MB. A * indicates timeout after 1 hr. Best figures are highlighted.

increasing number of philosophers. This is because the third premise involves the composition of (the negations of) all the environment assumptions.

The results are summarized in Figure 6. We observe that our A-G based approach is able to verify the system for up to 12 philosophers while SPIN times out after 9 philosophers. Furthermore the growth in time and space requirement is much better in our approach when compared to SPIN. The columns *T-Inc* and *M-Inc* show the ratio of time and memory between n and $n-1$ philosophers. This ratio increases monotonically for SPIN while it seems to be stabilizing (or increasing but at a much slower rate) for the A-G approach. This suggests that while increase in resource usage for verification cannot be entirely avoided with increasing number of components in the system, it can be expected to be more tractable if we use assume-guarantee reasoning.

7 Related Work

Throughout the paper we have already discussed related work on abstraction and assume-guarantee style verification. Here we discuss software model checking approaches that are most closely to ours.

SLAM [3] is a well-engineered tool that uses predicate abstraction, CEGAR and symbolic reachability analysis for verifying *sequential* C code. BLAST [6] has also focused primarily on CEGAR for sequential C code, using on-the-fly reachability for verification. Recently, BLAST has been extended to perform thread-modular abstraction refinement, in which assumptions and guarantees are both refined in an iterative fashion. The framework applies to concurrent programs that communicate through shared variables. The work in [16] also focuses on a shared-memory communicating programs but does not address notions of abstractions as is done in [22].

BANDERA [4] and JavaPathfinder (JPF) [33] are both aimed at verifying concurrent Java programs. BANDERA employs data abstraction and modular reasoning with user-supplied assumptions, but not automated assumption generation and CEGAR, while JPF is an explicit state model checker for the non-compositional analysis of Java code. VERISOFT [18] and FEAVER [24] perform verification on concurrent C code; FEAVER supports data abstraction, but no modular reasoning, while VERISOFT uses partial order reduction techniques for efficient state-less verification.

Learning in the context of software model checking has also been investigated in [20] and [25], but with a different goal. In those works, the L* Algorithm is used to generate models of software systems which can then be analyzed with model checking and testing techniques.

8 Conclusions

We presented a novel framework for performing abstraction and assume-guarantee reasoning for concurrent C code in an incremental and fully automated fashion. We discussed instantiations of the framework with symmetric and non-symmetric rules for the verification of systems with two or more components. We presented an implementation and case studies showing the merits of our approach. We evaluated the scalability of our approach with increasing number of components by comparing it to the state of the art model checker SPIN.

To evaluate how useful our approach is in practice, we are planning its extensive application to other (real-life) systems. However, our early experiments provide strong evidence in favor of this line of research. In the future, we plan to incorporate and evaluate other assume guarantee rules in our framework. We also plan to extend the types of properties that can be handled in our framework to include liveness and time related properties.

References

1. R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proc. of 10th CAV*, 1998.
2. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, Nov. 1987.
3. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. of PLDI*, June 2001.
4. BANDERA website. `http://bandera.projects.cis.ksu.edu/`.
5. H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu. Proof rules for automated compositional verification through learning. In *Proc. SAVCBS Workshop*, 2003.
6. BLAST website. `http://www-cad.eecs.berkeley.edu/~tah/blast/`.
7. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE TSE*, 30(6):388–402, June 2004.
8. S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *FMSD*, 2004.
9. S. Chaki, E. Clarke, N. Sharygina, and N. Sinha. Automated component substitutability analysis. *submitted.*
10. S. Chaki, J. Ouaknine, K. Yorav, and E. M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *Proc. of the 2nd Workshop on Software Model Checking (SoftMC '03)*, July 2003.

11. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV*, 2000.
12. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 2000.
13. E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proc. of the Fourth Symp. on Logic in Comp. Sci.*, pages 353–362, June 1989.
14. J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proc. of 9th TACAS*, pages 331–346, Apr. 2003.
15. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE'00*.
16. C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proc. of 10th SPIN*, pages 213–224, May 2003.
17. D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *Proc. of 17th ASE*, Sept. 2002.
18. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. of the* 24^{th} *ACM Symp. on Principles of Prog. Lang.*, pages 174–186, Jan. 1997.
19. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. of the Ninth Int. Conf. on Comp.-Aided Verification (CAV)*, pages 72–83, June 1997.
20. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Proc. of 8th TACAS*, pages 357–370, Apr. 2002.
21. O. Grumberg and D. E. Long. Model checking and modular verification. In *Proc. of the Second Int. Conf. on Concurrency Theory*, pages 250–265, Aug. 1991.
22. T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proc. of 15th CAV*, pages 262–274, July 2003.
23. T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc of 10th CAV*, pages 440–451, June, 1998.
24. G. J. Holzmann and M. H. Smith. Software model checking: Extracting verification models from source code. In *Proc. of FORTE/PSTV*, pages 481–497, Oct. 1999.
25. H. Hungar, O. Niese, and B. Steffen. Domain-specific optimizations in automata learning. In *Proc. of the 15th CAV*, 2003.
26. J. Ivers and N. Sharygina. Overview of ComFoRT: A model checking reasoning framework. *CMU/SEI-2004-TN-018*, 2004.
27. C. B. Jones. Specification and design of (parallel) programs. In *Information Processing 83: Proceedings of the IFIP 9th World Congress*, 1983.
28. Y. Kesten and A. Pnueli. Modularization and abstraction: The keys to practical formal verification. In *Proc. of MFCS*, 1998.
29. J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.
30. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logic and Models of Concurrent Systems*, volume 13, 1984.
31. R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, Apr. 1993.
32. SPIN website. `http://www.spinroot.com/`.
33. W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *Proc. of the Fifteenth IEEE Int. Conf. on Auto. Soft. Eng.*, pages 3–12, Sept. 2000.

Appendix

We provide here the proofs for the propositions that appear in the article.

Lemma 1. *For trace t and alphabets Σ, Σ': $\Sigma \subseteq \Sigma' \Rightarrow (t{\restriction}\Sigma'){\restriction}\Sigma = t{\restriction}\Sigma$.*

Lemma 2. *The following is true about the language of* $M_1 \parallel M_2$*:*

$$\mathcal{L}(M_1 \parallel M_2) = \{t \in (\alpha M_1 \cup \alpha M_2)^* \mid t{\upharpoonright}\alpha M_1 \in \mathcal{L}(M_1) \wedge t{\upharpoonright}\alpha M_2 \in \mathcal{L}(M_2)\}$$

Lemma 3. *Let* $t,\hat{t}$ *be traces,* Σ *be an alphabet, and* M *an FSM such that* $\hat{t} \in \mathcal{L}(M||[t,\Sigma])$*. Then for any alphabet* $\Sigma' \subseteq \Sigma$, $\hat{t}{\upharpoonright}\Sigma' = t{\upharpoonright}\Sigma$.

Proof. Follows from Proposition 2 and the fact that the alphabet of $[t,\Sigma]$ *is* Σ.

We are now ready to give the proofs. Here is the proof for Proposition 1, which states that Rule 2 is sound.

Proof. By contradiction: assume that the three premises hold but the conclusion of the rule does not hold. Consider a trace $t \in \mathcal{L}(M_1||M_2)$ that violates P, i.e. $t{\upharpoonright}\alpha P \notin \mathcal{L}(P)$. Let $\hat{t} = t{\upharpoonright}\alpha A$. Since $\alpha P \subseteq \alpha A$, it follows from Lemma 1 that $\hat{t}{\upharpoonright}\alpha P = t{\upharpoonright}\alpha P$, hence $\hat{t}{\upharpoonright}\alpha P \notin \mathcal{L}(P)$.

Moreover, $(t{\upharpoonright}(\alpha M_1 \cup \alpha A)){\upharpoonright}\alpha P$ is also the same as $t{\upharpoonright}\alpha P$ (and not in $\mathcal{L}(P)$), since $\alpha P \subseteq (\alpha M_1 \cup \alpha A)$ (from Lemma 1). From Lemma 2, $t{\upharpoonright}M_1 \in \mathcal{L}(M_1)$. Hence, by premise 1 and since $(t{\upharpoonright}(\alpha M_1 \cup \alpha A)){\upharpoonright}\alpha P \notin \mathcal{L}(P)$, it follows that $\hat{t} \notin \mathcal{L}(A_{M_1})$, i.e. $\hat{t} \in \mathcal{L}\left(\overline{A_{M_1}}\right)$. Similarly, by premise 2, $\hat{t} \in \mathcal{L}\left(\overline{A_{M_2}}\right)$. Since the alphabet of $\overline{A_{M_1}}||\overline{A_{M_2}}$ is also αA, it follows that $\hat{t} \in \mathcal{L}\left(\overline{A_{M_1}}||\overline{A_{M_2}}\right)$. Since $\hat{t}{\upharpoonright}\alpha P \notin \mathcal{L}(P)$, then $\overline{A_{M_1}}||\overline{A_{M_2}} \not\models P$, which contradicts premise 3.

Here is the proof for Proposition 2, which asserts the correctness of the counterexample analysis.

Proof. From Lemma 3, $\hat{t}{\upharpoonright}\alpha M_1 = t_1{\upharpoonright}\alpha M_1 \in \mathcal{L}(M_1)$. Similarly, $\hat{t}{\upharpoonright}\alpha M_2 \in \mathcal{L}(M_2)$. Therefore, from Lemma 2, $\hat{t} \in \mathcal{L}(M_1||M_2)$. From Lemma 3, $\hat{t}{\upharpoonright}\alpha P = t_1{\upharpoonright}\alpha P$. Hence $\hat{t}{\upharpoonright}\alpha P \notin \mathcal{L}(P)$.