A Counter Example Guided Abstraction Refinement Framework for Compositional Verification of Concurrent C Programs

Sagar Chaki, CMU

December 9, 2002

Abstract

Automatic verification of software implementations is a major challenge in the domain of formal methods. The state of the art solutions to this problem suffer from one or more of the following drawbacks. First, most tools attempt to scale to large implementations. But since they use trace containment as a notion of conformance, they risk an exponential blowup in the size of the specification. This potentially prevents them from handling examples where the specifications are large and complex. Second, there is little support for compositional reasoning. Third, abstraction refinement is performed either manually or by considering one counter example at a time. Multiple counter examples are not used simultaneously even though this could lead to better refinement schemes. Finally, counter example guided abstraction refinement is not integrated smoothly with the handling of concurrency. In this proposal I present a methodology that attempts to overcome all of these hurdles. First, exponential blowup due to complex specifications is avoided by using weak simulation as a notion of conformance. Second, compositional analysis is allowed for naturally by letting individual procedures be verified against their respective specifications. Third, during each abstraction refinement step, multiple counter examples are used to obtain a *minimal* set of predicates that suffices to refine the abstraction. Last, a *two-level abstraction refinement* scheme achieves smooth integration of concurrency with counter example guided abstraction refinement. In conjunction, these techniques are expected to enable verification of concurrent C programs against complicated specifications in an automated manner.

Contents

1 Motivation

The ability to reason about the correctness of programs is no longer a subject of primarily academic interest. With each passing day the complexity of software artifacts being produced and employed is increasing dramatically. There is hardly any aspect of our day-to-day lives where software agents do not play an often silent yet crucial role. The fact that many of such roles are safety-critical mandates that these software artifacts be validated rigorously before deployment. So far, however, this goal has largely eluded us.

Several factors contribute to this undesirable state of affairs. First, the sheer complexity of software. Binaries obtained from hundreds of thousands of lines of source code are routinely executed. The source code is written in languages ranging from C/C++/Java to ML/Ocaml. These languages differ not only in their flavor (imperative, functional) but also in their constructs (procedures, objects, pattern-matching, dynamic memory allocation, garbage collection), semantics (loose, rigorous) and so on.

This *sequential* complexity is but one face of the coin. Matters are further exacerbated by what can be called *parallel* complexity. State of the art software agents rarely operate in isolation. Usually they communicate and cooperate with other agents while performing their tasks. With the advent of the Internet, and the advance in networking technology, the scope of such communication could range from multiple threads communicating via shared memory on the same computer to servers and clients communicating via SSL channels across the Atlantic. Verifying the correctness of such complex behavior is a daunting challenge.

Another, much less visible yet important, factor is the development process employed in the production of most software and the role played by validation and testing methodologies in such processes. A typical instance of a software development cycle consists of five phases - (i) requirement specification, (ii) design, (iii) design validation, (iv) implementation and (v) implementation validation. The idea is that defects found in the design (in phase iii) are used to improve the design and those found in the implementation (in phase v) are used to improve the implementation. The cycle is repeated until each stage concludes successfully.

Usually the design is described using a formal notation like UML. The dynamic behavior is often described using Statecharts (or some variant of it.) The design validation is done by some exhaustive technique (like model checking.) However, what matters in the final tally is not so much the correctness of the design but rather the correctness of the implementation. Nevertheless, in reality, verification of the implementation is done much less rigorously. This makes it imperative that we focus more on developing techniques that enable us to verify actual code that will be compiled and executed. A major fraction of such code has been written, continues to be written and, in my opinion, will continue to be written in C.

Present day code validation falls in two broad categories - testing and formal verification. The merits and demerits of testing are well-known and thus it is unnecessary to dwell on them in detail here. It suffices to mention that the necessity of being *certain* about the correctness of a piece of code precludes exclusive reliance on testing as the validation methodology, and forces us to adopt more formal approaches. In summary, the demand for better formal techniques to verify concurrent and distributed C programs is currently overwhelming. This proposal attempts to provide a road map to a solution.

2 Software Verification: Past and Present

State of the art formal *software verification* is an extremely amorphous entity. Originally, most approaches in this field could be categorized as belonging to either of two schools of thought: theorem proving and model checking. In theorem proving, one attempts to construct a formula ϕ (in some suitable logic like higher-order predicate calculus) that represents both the system to be verified and the correctness property to be established. The validity of ϕ is then established using a theorem prover. As can be imagined, this approach is extremely powerful and can be used to verify virtually any imaginable system and property. The flip-side is that it involves a lot of manual effort. Furthermore it yields practically no diagnostic feedback that can be used for debugging if ϕ is found to be invalid.

Model Checking. Where theorem proving fails, model checking [18] shines. In this approach, the system to be verified is represented by a finite state transition system $\mathcal{M}$ (often a Kripke structure) and the property to be established is expressed as a temporal logic [31] (usually CTL with fairness or LTL) formula ϕ. The model checking problem is then to decide whether $\mathcal{M}$ is a model of ϕ. Not only can this process be automated to a large degree, it also yields extremely useful diagnostic feedback (often in the form of counter examples) if $\mathcal{M}$ is found not to model ϕ. Owing to these and other factors, the last couple of decades have witnessed the emergence of model checking as the eminent formal verification technique. Various kinds of temporal logics have been extensively studied [22] and efficient model checking algorithms have been designed [15, 39]. The development of techniques like *symbolic* model checking [28], *bounded* model checking [11], *compositional* reasoning [14] and *abstraction* [17] have enabled us to verify systems with enormous state spaces [29].

One of the original motivations behind the development of model checking was to extract and verify *synchronization skeletons* of concurrent programs, a typical software verification challenge. Somewhat ironically, the meteoric rise of model checking to fame was largely propelled by its tremendous impact on the field of hardware verification. I would argue that a major factor behind this phenomenon is that model checking can only be used if a finite model of the system is available. Also since real system descriptions are often quite large, the models must be extracted (semi)automatically. While this process is often straightforward for hardware, it is much more involved for software. Typically software systems have *infinite* state spaces. Thus extracting a finite model often involves a process of abstraction as well.

Predicate Abstraction. For a long time, the applicability of model checking to software was somewhat handicapped by the absence of powerful automated model extraction techniques. This scenario changed with the advent of predicate abstraction [40] (a related notion called data type abstraction used by systems like Bandera [1, 21] can be viewed as a special instance of predicate abstraction.) Even though predicate abstraction was quickly picked up for research in hardware verification as well [19, 20], its effect on code verification was rather dramatic. It forms the backbone of two of the major code verifiers in existence, SLAM [5, 9] and BLAST [2, 26].

Predicates abstraction is parameterized by a set of predicates involving the variables of the concrete system description. It also involves non-trivial use of theorem provers (in fact the its original use [40] was to create abstract state transition graphs using the theorem prover PVS.) Thus it has triggered a more subtle effect - it has caused the boundary between model checking and theorem proving to become less distinct. I believe this is an extremely promising trend and I plan to investigate other applications of theorem provers to facilitate the extraction of more precise models from software.

Counter Example Guided Abstraction Refinement. Even with progress in automated model extraction techniques, verifying large software systems remains an extremely tedious task. A major obstacle is created by the abstraction that happens during model extraction. Abstraction usually introduces additional behavior that is absent in the concrete system. Suppose that the model check fails and the model checker returns a counter example $\mathcal{C}$. This does not automatically indicate a bug in the system because it is entirely possible that $\mathcal{C}$ is an additional behavior introduced by abstraction (such a $\mathcal{C}$ is often called a spurious counter example.) Thus we need to verify whether $\mathcal{C}$ is spurious and if so we need to refine our model so that it no longer allows $\mathcal{C}$ as an admissible behavior. This process is called abstraction refinement. Since the extracted models and counter examples generated are quite large, abstraction refinement must be (semi)automated to be practically effective.

The above requirements lead naturally to the paradigm called counter example guided abstraction refinement (CEGAR). In this approach, the entire verification process is captured by a three step *abstract-verify-refine* loop. The actual details of each step depend on the kind of abstraction and refinement methods being used. The steps are described below in the context of predicate abstraction, where $\mathcal{P}$ denotes the set of predicates being used for the abstraction.

1. **Step 1 : Model Creation.** Extract a finite model from the code using predicate abstraction with $\mathcal{P}$ and go to step 2.
2. **Step 2 : Verification.** Check whether the model satisfies the desired property. If this is the case, the verification successfully terminates; otherwise, extract a counter example $\mathcal{C}$ and go to step 3.
3. **Step 3 : Refinement.** Check if $\mathcal{C}$ is spurious. If not we have an actual bug and the verification terminates unsuccessfully. Otherwise we improve $\mathcal{P}$ so that $\mathcal{C}$ and all previous spurious counter examples will be inadmissible if the model is extracted using the new $\mathcal{P}$ and go to step 1.

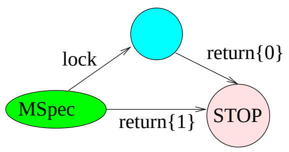
The methodology I propose will follow the CEGAR paradigm and will use predicate abstraction as *one of the* abstraction techniques. Two (and to the best of my knowledge, only two) other projects, SLAM [5] and BLAST [2], have similar goals and approaches. The natural question that arises is: how does my proposal relate to them? In the following sections I will layout the salient features and intended contributions of my proposal. I will also contrast and compare my proposal with related projects, and in particular with SLAM and BLAST.

3 LTS as Model and Specification

The first salient feature of my proposal is the uniform use of labeled transition systems (LTSs) to represent both the model of the system being verified and the property being checked.

LTS. Formally, a labeled transition system M is a 4-tuple (S, S_0, Act, T), where (i) S is a finite non-empty set of states, (ii) $S_0 \subseteq S$ is the set of initial states, (iii) Act is the set of actions, and (iv) $T \subseteq S \times Act \times S$ is the transition relation. We assume the presence of a distinguished action in the set Act, which we denote by ϵ. If $(s, a, s') \in T$, then (s, s') will be referred to as a a-transition and will be denoted by $s \xrightarrow{a} s'$. If s is reachable from s' via zero or more ϵ-transitions, we will denote this by $s \xrightarrow{\epsilon^\star} s'$. The relation $\Rightarrow$ is defined as follows: $s \overset{a}{\Rightarrow} s'$ iff there exist s_1 and s_2 such that $s \xrightarrow{\epsilon^\star} s_1 \xrightarrow{a} s_2 \xrightarrow{\epsilon^\star} s'$.

Action. Actions are atomic, and are distinguished simply by their names. Often, the presence of an action indicates a certain behavior which is achieved



Spec State Machine

```
int proc()
{
  if(do_lock())
    return 0;
  else
    return 1;
}
```

Code to be analysed

Figure 1: The example $\mathcal{S}$ and *proc*.

by a library routine in the implementation. Since we shall analyze a procedural language, namely C, we will model the termination of a procedure (i.e., a return from the procedure) by a special class of actions called *return actions*. Every return action a is associated with a unique return value $RetVal(a)$. Return values are either integers or *void*. All actions which are not return actions are called *basic* actions. A simple example of an LTS is shown in the left part of Figure 1.

From a theoretical point of view the presence of actions does not increase the expressive power of LTS and other related formalisms have also been investigated and used in this context. For example, SLAM uses Boolean programs [8] to model software while BLAST uses a variant of LTSs where the transitions are labeled with basic blocks and assume predicates rather than actions. It is my belief, however, that it is more natural for designers and software engineers to express the desired behavior of systems using a combination of states and actions. For example, the fact that a lock has been acquired or released can be expressed naturally by *lock* and *unlock* actions. In the absence of actions, the natural alternative is to introduce a new variable indicating the status of the lock, and update it accordingly. The LTS approach certainly is more intuitive, and allows both for a simpler theory and for an easier specification process.

C Implementation. In practice C programs can be quite complicated. However all non-recursive C programs can be inlined, resulting in a single monolithic *main* procedure. Therefore, if we ignore recursion, it is not unnaturally restrictive to assume that the C implementation consists of a single procedure. Using a single procedure as the *unit* of implementation also leads naturally to a compositional specification and verification methodology as we shall see soon. Recursion will be considered separately later. In the rest of the proposal I shall denote by *proc* the name of the implementation procedure.

Procedure Abstraction. As already mentioned, the specification of a procedure is expressed using a LTS. Intuitively the behaviors of the specification LTS represent the *acceptable* behaviors of the procedure. Thus the verification task is to show that the *possible* behaviors of the procedure are also behaviors of the specification. In practice, it often happens that a single procedure performs quite different tasks for certain settings of its parameters. In my methodology, this phenomenon is accounted for by allowing multiple LTSs to represent a single procedure. The selection among these LTSs is achieved by *guards*, i.e., formulas, which describe the conditions on the procedure parameters under which a certain LTS is applicable. This gives rise to the notion of *procedure abstraction* (PA); formally a PA for a procedure *proc* is a tuple $\langle d, l \rangle$ where

1. d is the declaration for *proc*, as it appears in a C header file.
2. l is a finite list $\langle g_1, M_1 \rangle, \ldots, \langle g_n, M_n \rangle$ where each g_i is a guard formula ranging over the parameters of *proc*, and each M_i is an LTS with a single initial state.

The procedure abstraction expresses that *proc* conforms to one LTS chosen among the L_i's. More precisely, *proc* conforms to L_i if the corresponding guard g_i evaluates to true over the *actual arguments* passed to *proc*. The guard formulas g_i are required to be mutually exclusive and complete so that the choice of L_i is unambiguous and always well-defined. The role of PAs in this process is twofold:

1. A *target PA* is used to describe the desired behavior of the procedure *proc*.
2. To assist the verification process, we employ valid PAs (called the *assumption* PAs) for library routines used by *proc*.

Thus, PAs can be seen both as conclusions and as assumptions of the verification process. Consequently, the proposed methodology yields a scalable and compositional approach for verifying large software systems. I believe that the form of compositionality provided by my approach is unique among existing software verification systems. In particular, such compositionality is absent in SLAM and BLAST. In addition both these projects use transition systems without actions to express specifications. Figure 1 describes a simple case of a procedure *proc* and a corresponding LTS.

4 Simulation as Conformance

The fundamental goal of a software verifier is to prove that the behavior of an implementation *conforms* to that of its specification. The crucial decision is therefore: what is an appropriate notion of conformance between two systems. Most existing tools (including SLAM and BLAST) have opted for *trace containment* as the notion of conformance. This means that an implementation $\mathcal{I}$ conforms to a specification $\mathcal{S}$ iff every trace of $\mathcal{I}$ is also a trace of $\mathcal{S}$. This choice has an important ramification. Normally, checking trace containment between $\mathcal{I}$ and $\mathcal{S}$ is done in two steps:

1. Construct a system $\mathcal{S}'$ such that the traces of $\mathcal{S}'$ are exactly the ones that are not traces of $\mathcal{S}$.
2. Check that no trace of $\mathcal{I}$ is a trace of $\mathcal{S}'$.

The problem is that if $\mathcal{S}$ is allowed to be non-deterministic, the size of $\mathcal{S}'$ could in general be exponential in the size of $\mathcal{S}$. Note that we are implicitly assuming that $\mathcal{I}$ and $\mathcal{S}$ are expressed as some kind of finite state transition system. This problem is more critical if $\mathcal{S}$ is relatively large, as is invariably the case with software of even moderate complexity. Thus the necessity of verifying large complex specifications mandates that we find a way of avoiding this exponential blowup. In my methodology, this is achieved by using a stronger notion of conformance viz. *weak simulation*.

Weak Simulation. Let $M = (S, S_0, Act, T)$ and $M' = (S', S'_0, Act, T')$ be two LTSs. A relation $E \subseteq S \times S'$ is called a *weak simulation* between M and M' iff (i) for all $s \in S_0$ there exists $s' \in S'_0$ such that $(s, s') \in E$, (ii) $(s, s') \in E$ implies that for all actions $a \in Act \setminus \{\epsilon\}$ if $s \stackrel{a}{\Rightarrow} s_1$, then there exists $s'_1 \in S'$ such that $s' \stackrel{a}{\Rightarrow} s'_1$ and $(s_1, s'_1) \in E$ and (iii) $(s, s') \in E$ implies that if $s \stackrel{\epsilon}{\Rightarrow} s_1$, then either $(s_1, s') \in E$ or there exists $s'_1 \in S'$ such that $s' \stackrel{\epsilon}{\Rightarrow} s'_1$ and $(s_1, s'_1) \in E$. We say that LTS M' *weakly simulates* M (denoted by $M \sqsubseteq M'$) if there exists a weak simulation relation $E \subseteq S \times S'$ between M and M'. In the rest of this proposal, I shall use the convention that the terms *simulation* and *simulates* will always mean *weak simulation* and *weakly simulates* respectively.

There exist well known *polynomial* time algorithms for checking simulation between two LTSs. In particular there exists a reduction of this problem to that of satisfiability of N-HORNSAT formulas based on [41]. There also exists efficient online linear time N-HORNSAT satisfiability algorithm based on [7]. This enables us to check simulation between two LTSs very efficiently, putting complex implementations and specifications within our grasp.

There is a small piece of the puzzle that is still not in place. Suppose $\mathcal{I}$ is the finite model extracted from *proc* using predicate abstraction. The existence of a simulation relation between $\mathcal{I}$ and $\mathcal{S}$ does not immediately enable us to conclude that *proc* is correct. However if we are able to show that $\mathcal{I}$ simulates *proc*, then by the transitive property of simulation we would also be able to conclude that $\mathcal{S}$ simulates *proc*. Then the correctness of *proc* would follow immediately. Thus the general problem to be solved is summarized in the following theorem.

Theorem 1 *If an LTS $\mathcal{I}$ is extracted from an implementation proc using predicate abstraction, then $\mathcal{I}$ simulates proc.*

I propose to prove Theorem 1 as part of my project. To the best of my knowledge this is an entirely possible yet unproved result. Projects like SLAM and BLAST have already shown similar results in the context of trace containment since that is their notion of conformance. It should be noted however that simulation is a stronger notion than trace containment. Hence Theorem 1 would immediately imply the following corollary.

Corollary 1 *If an LTS $\mathcal{I}$ is extracted from an implementation proc using predicate abstraction, then $\mathcal{I}$ trace contains proc.*

5 CEGAR for Simulation

In this section I shall describe my proposal to incorporate CEGAR into the simulation based framework I have built up so far. There are four critical components of this proposal: (i) what is a counter-example when the notion of conformance between two systems is simulation, (ii) given two systems that are not in conformance how can we construct the counter-example efficiently, (iii) how can we check efficiently whether the counter-example is valid or spurious and (iv) how can we refine $\mathcal{I}$ such that the counter-example can no longer reappear in subsequent simulation checks. To the best of my knowledge these questions have not been investigated before. Once again, projects like SLAM [10] and BLAST have not concerned themselves with these problems simply because they do not use simulation. I discuss some possible solutions in the following sections. Obviously these are starting points. A final fully satisfactory solution will require further development.

5.1 Counter-Example

Let $M = (S, S_0, Act, T)$ and $M' = (S', S'_0, Act, T')$ be two LTSs such that M' does not simulate M. When the notion of conformance is trace containment a counter-example is simply a trace of M that is not a trace of M'. However with simulation a counter-example can be viewed as a directed acyclic graph with edges labeled by actions (i.e. a directed acyclic LTS), which we call the CE-DAG. Before we describe the CE-DAG in greater detail we need to discuss a few other issues.

The standard iterative algorithm for computing the largest simulation relation between M and M' proceeds as follows. It starts with $S \times S'$ as the initial value of the result and then iteratively refines it (i.e. removes elements from it) till a fixed point is reached. Even though the final result (the greatest fixed point, which we denote by Σ) is guaranteed to be unique, the order in which elements are removed may vary depending on the actual implementation details of the algorithm. However if we select a particular execution of the algorithm it automatically induces a strict ordering on the elements of $(S \times S') \setminus \Sigma$ viz. $x > y$ iff x was removed after y during the execution. Let Γ be one such ordering. The significance of this paragraph is that even though we do not use the standard iterative algorithm to compute Σ, our N-HORNSAT solver can yield such an ordering Γ, and this ordering is all we need to construct a CE-DAG.

We are now ready to describe the CE-DAG. The set of nodes of the CE-DAG is $(S \times S') \setminus \Sigma$. Note that since M is not simulated by M', (S_0, S'_0) must be a node of the CE-DAG. The edges of the CE-DAG are determined as follows. There is an edge from (s, s') to (t, t') labeled with a iff the following conditions hold:

1. $(s, s') > (t, t')$ according to the ordering Γ.
2. $(s, a, t) \in T$ and if X is the set of states $\{x \in S' \mid (s', a, x) \in T'\}$ then (i) $t' \in X$ and (ii) for every $x \in X$, $(t, x) \in (S \times S') \setminus \Sigma$.

Thus the edges of the CE-DAG capture causality between the exclusion of its nodes from Σ. In particular, an edge from x to y labeled with a indicates that the exclusion of x from Σ is a direct consequence of the exclusion of y from Σ and the execution of action a from x. Without loss of generality we may assume that the element (S_0, S'_0) is the last in the ordering Γ (i.e. the last to be excluded from Σ) and is also the unique source node (has only outgoing edges) of the CE-DAG. Given Γ, we can construct the CE-DAG efficiently using polynomial time in $|S| \times |S'|$. I shall not go into the details of the exact algorithm for doing this now.

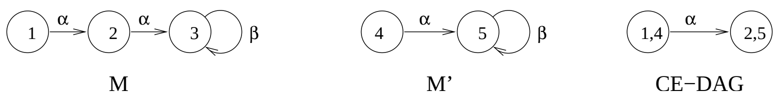


Figure 2: Two example LTSs and a CE-DAG.

Example 1 *Consider the LTSs M and M' in Figure 2. A possible ordering Γ in this case is the following:* $(3,4) < (2,5) < (1,4)$*. The CE-DAG has two relevant nodes* $(1,4)$ *and* $(2,5)$ *with a single edge from* $(1,4)$ *to* $(2,5)$ *labeled with* α*. It is shown in Figure 2.*

5.2 Validity of a CE-DAG

Since a CE-DAG is a finite DAG it has only a finite number of paths. Each such path p gives rise to a corresponding concrete execution path $\gamma(p)$ in the actual C program. Given p one can efficiently construct $\gamma(p)$ and then check its validity i.e. whether $\gamma(p)$ is actually possible in any execution of the C program. The CE-DAG is said to be a valid counter-example iff for each path p in it, $\gamma(p)$ is a valid concrete execution path. Conversely the CE-DAG is a spurious counter-example if there is a path p in it such that $\gamma(p)$ is spurious.

There are two approaches to checking the validity of a concrete path $\gamma(p)$ - the forward approach based on the computation of strongest post-conditions and the backward approach based on the computation of weakest preconditions. Let $\gamma(p)$ be the sequence of statements $s_1, \ldots, s_n$. In both approaches to checking the validity of $\gamma(p)$ we compute a set of expressions e_i for $1 \leq i \leq (n+1)$. In the forward approach we start with $e_1 = true$ and let e_{i+1} be the strongest post-condition of e_i w.r.t. s_i. Then $\gamma(p)$ is valid iff we can compute up to e_{n+1} without getting false. In the backward approach we start with $e_{n+1} = true$ and let e_i be the weakest precondition of e_{i+1} w.r.t. s_i. Then $\gamma(p)$ is valid iff we can compute up to e_1 without getting false.

If $\gamma(p)$ is spurious, by combining the forward and backward approaches we can compute a *minimal spurious sub-path* of $\gamma(p)$ as follows. We start with the forward approach and keep computing the e_is. Let e_{q+1} be the first time we get false. Now starting with s_q and with $e_{q+1} = true$ we apply the backward approach. Let e_p be the first time we get false. Then $s_p, \ldots, s_q$ is the minimal spurious sub-path of $\gamma(p)$. Since multiple paths in the CE-DAG might share common spurious sub-paths, by using the minimal spurious sub-paths judiciously we hope to avoid significant redundant computation. Note that a similar notion of minimal spurious sub-path has been used in the context of hardware verification in [19].

5.3 Refining the Abstraction

We have arrived at the last of the four components of CEGAR viz. how to improve our abstraction $\mathcal{I}$ such that all the previously discovered spurious CE-DAGs do not reappear in subsequent trials. In general each CE-DAG δ is spurious because it contains a path p such that $\gamma(p)$ is spurious. We know that if we add the e_is discovered while checking $\gamma(p)$'s validity as predicates and recompute $\mathcal{I}$, then $\gamma(p)$ (and hence δ) can no longer reappear. However the problem with this approach is the large number of predicates it might yield. In particular, there might be several spurious CE-DAGs, each CE-DAG will have a spurious path and each spurious path could yield several predicates. The problem is particularly grave since we already know that the construction of $\mathcal{I}$ consumes exponential resources in the number of predicates involved.

Thus it would be extremely desirable to be able to solve the following problem: given a set of spurious CE-DAGs $\Delta = \{\delta_1, \ldots, \delta_k\}$, construct a *minimal* set of predicates $\mathcal{P}$ such that the $\mathcal{I}$ constructed using $\mathcal{P}$ may not yield any element of Δ as a counter-example. We now present a solution to this problem. Let p_i be the spurious path in δ_i and let P_i be the set of e_is discovered while checking for the validity of $\gamma(p_i)$. We know that the set $P = \cup_{i=1}^{k} P_i$ will definitely eliminate each δ_i as a CE-DAG. However our goal is to come up with a smaller subset of P that will also suffice in this regard. We achieve this by reducing the problem to an instance of pseudo Boolean satisfiability. A pseudo Boolean formula is a standard Boolean formula together with a minimization criterion. The reduction is done as follows.

First, for each predicate $R_i \in P$ we introduce a Boolean variable B_i. Then for each spurious path p_i let C_i be a set of subsets of P with the following property: $C_i = \{Q_1, \ldots, Q_{n_i}\}$ such that if we use any Q_j as the set of predicates then the resulting $\mathcal{I}$ does not have p_i as a possible execution trace. For example suppose $C_1 = \{\{R_1\}, \{R_2, R_3\}\}$. This means that if we use either R_1 or use R_2 and R_3 as predicates, the resulting $\mathcal{I}$ will not contain p_1 as a path. Given p_i it is possible to compute a suitable C_i by trying out various possible combinations of R_i and checking which of these combinations are capable of eliminating p_i. This process could be exponential in the number of predicates in the worst case but could be speeded up in practice using various heuristics.

Suppose $Q_i = \{R_{\alpha_1}, \ldots, R_{\alpha_m}\}$. Then let $\phi(Q_i)$ be the Boolean conjunctive clause $\wedge_{i=1}^{m} B_{\alpha_i}$. Then for $C_i = \{Q_1, \ldots, Q_{n_i}\}$ let $\psi(C_i)$ be the Boolean formula $\vee_{j=1}^{n_i} \phi(Q_j)$. Finally the Boolean formula we wish to solve is $\wedge_{i=1}^{k} \psi(C_i)$. In addition we wish to minimize the following quantity: $\Sigma_{R_i \in P} B_i$. Once we get a solution to this pseudo Boolean formula we choose $\mathcal{P}$ to be the set of predicates

R_i such that B_i is assigned true by the solution.

Example 2 *Let* $\Delta = \{\delta_1, \delta_2\}$ *and* $P = \{R_1, R_2, R_3\}$. *Let* $C_1 = \{\{R_1\}, \{R_2, R_3\}\}$ *and* $C_2 = \{\{R_2, R_3\}\}$. *Then we get* $\psi(C_1) = (B_1) \vee (B_2 \wedge B_3)$ *and* $\psi(C_2) = B_2 \vee B_3$. *Thus the formula to be solved is* $\psi(C_1) \wedge \psi(C_2)$ *along with the minimization criterion* $(B_1 + B_2 + B_3)$. *It is obvious that assigning true to any two of the Boolean variables and false to the third gives a valid solution. For example one solution is* $\{B_1 = true, B_2 = true, B_3 = false\}$ *which yields* $\mathcal{P} = \{R_1, R_2\}$.

Here is the essential idea behind the above reduction. Each constraint $\phi(Q_i)$ guarantees that all the predicates in Q_i will be included in $\mathcal{P}$. Thus each $\psi(C_i)$ guarantees that all the predicates from at least one of the elements of C_i will be included in $\mathcal{P}$ and hence p_i will be eliminated as a trace. Therefore the entire formula ensures that every p_i will be eliminated as a trace. The minimization criterion guarantees that a minimal set of predicates which eliminate every p_i will be selected as $\mathcal{P}$. Note that instead of pseudo Boolean satisfiability we could also have used *logic minimization* or *hitting set* to solve this problem. However the major motivation behind using pseudo Boolean satisfiability is to leverage considerable recent progress in this area [6].

6 Concurrency

The addition of concurrency causes a quantum jump in the complexity of any verification procedure. The most apparent cause of this phenomenon is the exponential blowup in the state space due to the parallel composition of a number of concurrently executing components. Another subtle but significant blowup occurs in the number of possible execution interleavings if the components are allowed to perform their actions asynchronously. Both these problems plague hardware verification to a large degree. The blowup in the state space is tackled by techniques like abstraction and compositional reasoning [16, 35] while the blowup in the interleavings is handled by partial order reduction [24, 32, 37, 44] related approaches.

In the case of software, the above problems are further compounded by the diversity of communication mechanisms. A host of mechanisms have been proposed in the literature and used, ranging from shared memory and monitors to blocking and non-blocking message passing. In addition a wide variety of formalisms have been proposed for modeling concurrency and interaction. These range from the message passing based process algebras like CCS [33], CSP [27] and the Pi-calculus [34] to the rendezvous based Petri nets [38]. In my methodology I use asynchronous parallel composition with synchronization on shared actions as the communication mechanism. The choice is motivated by its clean semantics and ease of integration with a CEGAR formalism.

Parallel Composition. Let $M_1 = \langle S_1, S_{0,1}, Act_1, T_1 \rangle$ and $M_2 = \langle S_2, S_{0,2}, Act_2, T_2 \rangle$ be two LTSs. Then the asynchronous composition of M_1 and

M_2 (denoted by $M_1 \parallel M_2$) is the LTS $\langle S_\parallel, S_{0,\parallel}, Act_\parallel, T_\parallel \rangle$ where (i) $S_\parallel = S_1 \times S_2$, (ii) $S_{0,\parallel} = (S_{0,1}, S_{0,2})$, (iii) $Act_\parallel = Act_1 \cup Act_2$ and (iv) $((s_1, s_2), a, (t_1, t_2)) \in T_\parallel$ iff one of the following conditions hold:

1. $a \in Act_1 \setminus Act_2$ and $t_1 = t_2$ and $s_1 \xrightarrow{a} s_2$.
2. $a \in Act_2 \setminus Act_1$ and $s_1 = s_2$ and $t_1 \xrightarrow{a} t_2$.
3. $a \in Act_1 \cap Act_2$ and $s_1 \xrightarrow{a} s_2$ and $t_1 \xrightarrow{a} t_2$.

A nice property of the above composition semantics is the resulting validity of Theorem 2. I will use Theorem 2 in the following sections while presenting a two-level abstraction refinement framework that aims to integrate concurrency with CEGAR.

Theorem 2 (Parallel Composition and Simulation) *Let $P_1, \ldots, P_n$ and $Q_1, \ldots, Q_n$ be arbitrary LTSs. Then the following proof rule is valid.*

$$\frac{P_1 \sqsubseteq Q_1 \quad \ldots \quad P_n \sqsubseteq Q_n}{(P_1 \parallel \ldots \parallel P_n) \sqsubseteq (Q_1 \parallel \ldots \parallel Q_n)}$$

Several model checking based software verification tools have taken a shot at concurrency. Surprisingly most of these tools like Bandera [1, 21], Java PathFinder [4, 25] and Calvin [23] target Java programs. However to the best of my knowledge these tools do not incorporate automated abstraction refinement. On the other hand tools like SLAM and BLAST that do target automated abstraction refinement and C have no support for concurrency. I believe that my proposed methodology is unique in attempting to combine concurrency with automated abstraction refinement.

6.1 Two Level Abstraction Refinement

In my framework a concurrent C program is assumed to consist of a finite sequence of C procedures $proc_1, \ldots, proc_n$. The idea is that the execution of the program involves the concurrent execution of all these procedures. Then the two level CEGAR framework can be summarized by the following steps.

1. **Step 1: C Abstraction.** Obtain LTS models $\mathcal{I}_i$ by predicate abstraction of $proc_i$ for $1 \leq i \leq n$. Go to step 2.
2. **Step 2: LTS Abstraction.** Obtain an abstract LTS $\mathcal{A}_i$ be applying an *LTS abstraction* on $\mathcal{I}_i$ for $1 \leq i \leq n$. We denote the LTS abstraction algorithm by α_{LTS}. It must obey the property that $\mathcal{L} \sqsubseteq \alpha_{LTS}(\mathcal{L})$ for any LTS $\mathcal{L}$. Go to step 3.
3. **Step 3: Verification.** Check whether $(\mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n) \sqsubseteq \mathcal{S}$. If the answer is *yes* then by Theorem 1 and Theorem 2 we can conclude the verification successfully. Otherwise let $\mathcal{C}$ be a counter example obtained. Go to step 4. For simplicity I shall assume that $\mathcal{C}$ is a trace but the methodology works equally well for a CE-DAG.

4. **Step 4: LTS Abstraction refinement.** Let $\mathcal{C}_i$ be the *projection* of $\mathcal{C}$ on the actions of $\mathcal{A}_i$. For $1 \leq i \leq n$ we check if $\mathcal{C}_i$ is a possible path of $\mathcal{I}_i$. If this is the case for every i we proceed to step 5. Otherwise let k be the smallest index such that $\mathcal{C}_k$ is not a possible path of $\mathcal{I}_k$. We refine $\mathcal{A}_k$ such that $\mathcal{C}_k$ and all previous such spurious counter examples are no longer admissible in the new $\mathcal{A}_k$ and go to step 3. We denote this refinement procedure by ρ_{LTS}.
5. **Step 5:C Abstraction refinement.** For $1 \leq i \leq n$ we check if $\mathcal{C}_i$ is a possible path of $proc_i$. If this is the case for every i then we have found a real counter example and the verification terminates unsuccessfully. Otherwise let k be the smallest index such that $\mathcal{C}_k$ is not a possible path of $proc_k$. We refine $\mathcal{I}_k$ (using predicate abstraction refinement discussed earlier) such that $\mathcal{C}_k$ and all previous such spurious counter examples are no longer admissible in the new $\mathcal{I}_k$ and go to step 3.

The above algorithm captures the complete methodology I am proposing. One should note that this approach tries to avoid state explosion problem by only composing as small systems as possible. In other words it starts with very small components $\mathcal{A}_1, \ldots, \mathcal{A}_n$ and tries to prove or disprove the property using them. A component is refined only if it is found to be too abstract. Also refinement is done locally. In other words components are analyzed individually (steps 4 and 5) to decide if they need to be refined. In the case were refinement is necessary, it can also be done for a single component in isolation. The only step in which composition is necessary is the actual verification (step 3). I hope to leverage existing technology for coping with the blowup incurred in this phase.

Several of the steps in this methodology (more specifically 1, 3 and 5) have been discussed previously in the context of verifying purely sequential programs. The other steps involve developing effective abstraction (α_{LTS}) and refinement (ρ_{LTS}) schemes for LTSs. My research in this direction is still quite nascent but in the rest of this section I will try to briefly summarize some directions that I intend to pursue.

LTS Abstraction. The general scheme for LTS abstraction works as follows. Let $M = \langle S, S_0, Act, T\rangle$ be an arbitrary LTS. Then $\alpha_{LTS}(M)$ is an LTS $\langle S', S'_0, Act', T'\rangle$ such that:

- S' is a subset of 2^S such that every state in S belongs to some element of S' i.e. $\forall s \in S, \exists s' \in S'$ such that $s \in s'$.
- S'_0 consists of the subsets of S that contain at least one initial state of M. In other words $S'_0 = \{s' \in S' \mid \exists i \in S_0 \text{ such that } i \in s'\}$.
- $Act' = Act$.
- $T' = \{(s', a, t') \mid \exists s, t \in S \text{ such that } s \in s' \wedge t \in t' \wedge (s, a, t) \in T\}$.

The pleasant property of this abstraction scheme is that it automatically guarantees $M \sqsubseteq \alpha_{LTS}(M)$. Also it can be constructed efficiently from M. Furthermore this scheme is completely parameterized by the set of abstract states S'. In other words once S' is fixed the rest of the abstraction is uniquely determined. I plan to investigate several approaches to constructing S'. For example a promising approach is to construct S' by *clubbing together* states of S whose outgoing transitions are labeled by identical sets of actions. I also plan to investigate appropriate refinement schemes for such abstractions.

7 Milestones and Deadlines

I have already implemented a part of the proposed framework. My tool *MAGIC* [13] extracts LTS models from sequential C programs using predicate abstraction. It then checks simulation between the model and user supplied specifications. *MAGIC* accepts specifications described in the FSP [30] notation. As part of *MAGIC* I have implemented an online N-HORNSAT algorithm based on [7, 42]. I have experimented with code fragments from the Linux kernel and the OpenSSL toolkit and obtained encouraging results [13].

CEGAR. Currently I am implementing the CEGAR framework for simulation. I am also experimenting with various optimization strategies for extracting a minimal set of predicates for abstraction refinement. My goal is to be able to verify separately the client and servers components OpenSSL example completely automatically using CEGAR. I hope to achieve by the end of April 2003.

Concurrency. I also want to start implementing the proposed framework for handling concurrent C programs. There is tremendous scope for experimenting with a wide spectrum of abstractions for LTSs and I am hopeful about getting illuminating results from them. Once again I wish to use the OpenSSL example as the benchmark. The advantage is that the OpenSSL protocol implementation is inherently concurrent. It consists of code for both clients and servers. Thus, being able to prove properties of the system comprising of both a client and a server would be a perfect showcase for my proposed methodology. My aim is to complete these experiments by November 2003.

Symbolic Techniques. My goal is to complete and defend my thesis by the end of Spring 2004. Therefore, time permitting, I wish to look into a number of other related issues. First, the current implementation is explicit state in the sense that the transitions of the extracted model LTS are stored explicitly. This is quite expensive in practice. I wish to investigate the problem of whether these transitions can be computed and stored more efficiently using symbolic techniques like BDDs [12].

Recursion. So far I have ignored recursive C programs. Though I do not expect this to be a big problem, I would like to look into techniques for handling recursion as well. The obvious approach is as follows - when dealing with a recursive procedure use the same procedure abstraction both as a target and an assumption simultaneously. This appears to be cyclic and therefore potentially unsound. However I think this is still a sound approach and would like to obtain a formal proof of this claim.

Static Analysis. Predicate abstraction can often be aided by static analysis information. For example, the computation of precise weakest preconditions in the presence of pointers requires the availability of aliasing information. I would like to investigate this issue further and explore deeper connections between precise model extraction and static analysis. Part of my goal is also to interface my tool with efficient static analysis engines like CodeSurfer [3]. In addition, most static analysis problems can be solved by polynomial time algorithms. This suggests that there are efficient reductions of these problems to HORNSAT (which is P-complete.) Since I already have a N-HORNSAT engine implemented, finding such efficient reductions would let me integrate these analysis more tightly with *MAGIC*.

Proof Carrying Code. Another promising direction is to incorporate features of proof carrying code [36] in this framework. The motivation behind being able to generate short easily checkable proofs of conformance between a design and an implementation is most evident in security-related applications. For example, a web browser often downloads applets from untrusted sources and executes them locally. If a proof of conformance of this applet to some standard secure design was shipped along with the applet, then the web-browser could check the proof before loading and executing the applet. Several people have looked into the notion of proofs for model checking [43]. I am looking into this problem with respect to my notion of conformance. For instance the maximal simulation relation is a potential candidate for a proof.

8 Conclusion

The importance of being able to rigorously verify C programs, especially concurrent C programs, can be hardly overstated. Not only must we be able to handle large implementations, our techniques should be capable of scaling to large and complex specifications as well. Most existing tools focus on large implementations. But by opting for trace containment as a notion of conformance they are potentially incapable of handling large specifications. In addition, abstraction refinement is done by considering one counter example at a time and support for concurrency is rarely integrated smoothly into an automated CEGAR framework. This proposal attempts to overcome the above shortcomings. By using simulation as a notion of conformance it can scale to significantly large specifications. Compositional verification is enabled naturally by allowing the verification to be carried out on a per-procedure basis. During each abstraction refinement step multiple counter examples are used simultaneously to obtain a minimal set of predicates that will suffice to refine the abstraction. Finally a two-level abstraction refinement scheme achieves smooth integration of concurrency with CEGAR. I believe that if the major goals of my project materialize, it will result in a significant advancement of the state of the art in automated software verification technology.

References

[1] Bandera. *http://www.cis.ksu.edu/santos/bandera*.

[2] BLAST. *http://www-cad.eecs.berkeley.edu/ rupak/blast*.

[3] Codesurfer. *http://www.grammatech.com*.

[4] Java PathFinder. *http://ase.arc.nasa.gov/visser/jpf*.

[5] SLAM. *http://research.microsoft.com/slam*.

[6] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. PBS: A Backtrack-Search Psuedo-Boolean Solver and Optimizer. In *Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, 2002.

[7] Giorgio Ausiello and Giuseppe F. Italiano. On-line algorithms for polynomially solvable satisfiability problems. *Journal of Logic Programming*, 10(1,2,3 & 4):69–90, January 1991.

[8] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.

[9] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. *Lecture Notes in Computer Science*, 2057, 2001.

[10] Thomas Ball and Sriram K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research, Redmond, January 2002.

[11] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

[12] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[13] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *International Conference on Software Engineering (ICSE), To appear*, 2003.

[14] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 353–362. IEEE Press, 1989.

[15] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and System (TOPLAS)*, 8(2):244–263, April 1986.

[16] E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking algorithms. In *Proc. of the 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 294–303, 1987.

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

[18] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[19] Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Fourth International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2002.

[20] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *Computer Aided Verification*, pages 160–171, 1999.

[21] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Hongjun Zheng, and Willem Visser. Tool-supported program abstraction for finite-state verification. In *International Conference on Software engineering*, pages 177–187. IEEE Computer Society, 2001.

[22] E. A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 995–1072, 1990.

[23] C. Flanagan, S. Qadeer, and S. Seshia. A modular checker for multithreaded programs. In *Proceedings of Computer Aided Verification*, 2002.

[24] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of Computer Aided Verification*, pages 321–340, 1990.

[25] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[26] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[27] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM (CACM)*, 21(8):666–677, August 1978.

[28] J.R. Burch, E.M. Clarke, D.E. Long, K.L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[29] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[30] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. Wiley, 2000.

[31] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[32] K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design: An International Journal*, 6(1):45–65, January 1995.

[33] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[34] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[35] Kedar Sharad Namjoshi. *Ameliorating the State Space Explosion Problem*. PhD thesis, UT Austin, 1998.

[36] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[37] D. Peled. All from one, one for all: On model checking using representatives. In *Proceedings of Computer Aided Verification*, pages 409–423, 1993.

[38] Carl Adam Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. of IFIP Congress 62*, pages 386–390, Amsterdam, 1963. North Holland Publ. Comp.

[39] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, pages 337–351, 1982.

[40] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification*, volume 1254, pages 72–83. Springer Verlag, 1997.

[41] Sandeep K. Shukla. *Uniform Approaches to the Verification of Finite State Systems*. PhD thesis, SUNY, Albany, 1997.

[42] Sandeep K. Shukla, Harry B. Hunt III, and Daniel J. Rosenkrantz. HORN-SAT, model checking, verification and games. Technical Report TR-95-8, State University of New York, Albany, 1995.

[43] L. Tan and R. Cleaveland. Evidence-based model checking. In *Proceedings of Computer Aided Verification*, 2002.

[44] A. Valmari. A stubborn attack on state explosion. In *Proceedings of Computer Aided Verification*, pages 25–42, 1990.