Compositionality Entails Sequentializability

Pranav Garg and P. Madhusudan

University of Illinois at Urbana-Champaign

Abstract. We show that any concurrent program that is amenable to compositional reasoning can be effectively translated to a sequential program. More precisely, we give a reduction from the verification problem for concurrent programs against safety specifications to the verification of sequential programs against safety specifications, where the reduction is parameterized by a set of auxiliary variables A, such that the concurrent program *compositionally* satisfies its specification using auxiliary variables A iff the sequentialization satisfies its specification. Existing sequentializations for concurrent programs work only for underapproximations like bounded context-switching, while our sequentialization has the salient feature that it can *prove* concurrent programs entirely correct, as long as it has a compositional proof. The sequentialization allows us to use sequential verification tools (including deductive verification tools and predicate abstraction tools) to analyze and prove concurrent programs correct. We also report on our experience in the deductive verification of concurrent programs by proving their sequential counterparts using the program verifier Boogie.

Keywords: concurrent programs, compositional verification, sequentialization

1 Introduction

Sequentializing concurrent programs has been a topic of recent research. Given a concurrent program with a safety specification, we would like to reduce the problem of verifying the concurrent program to the verification of a sequential program. Moreover, and most importantly, we seek a sequential program that does not simply simulate the global evolution of the concurrent program as that would be quite complex and involve taking the product of the local state-spaces of the processes. Instead, we seek a sequential program that tracks a *bounded* number of copies of the local and shared variables, where the bound is independent of the number of parallel components.

The appeal of sequentialization is that it allows using the existing class of sequential verification tools to verify concurrent programs. A large number of sequential verification techniques and tools, like deductive verification, abstraction-based model-checking, and static dataflow analysis immediately come into play when a sequentialization is possible.

Of course, such sequentializations are not possible for all concurrent programs and specifications. In fact, in the presence of recursion and when variables have

bounded domains, concurrent verification is undecidable while sequential verification is decidable, which proves that an effective sequentialization is in general impossible.

The currently known sequentializations have hence focussed on capturing *under-approximations* of concurrent programs. Lal and Reps [13] showed that given a concurrent program with finitely many threads and a bound k, the problem of checking whether the concurrent program is safe on all executions that involve only k context-rounds can be reduced to the verification of a sequential program. A *lazy* sequentialization for bounded context rounds that ensures that the sequential program explores only states reachable by the concurrent program was defined by La Torre et al [10]. A sequentialization for unboundedly many threads and bounded round-robin rounds of context-switching is also known [11]. Lahiri, Qadeer and Rakamarić have used the sequentialization of Lal and Reps to check concurrent C-programs by unrolling loops in the sequential program a bounded number of times, and subjecting them to deductive SMT-solver based verification [12, 8].

In this paper, we show a general sequentializability result that is not restricted to under-approximations. We show that any concurrent program with finitely many threads can *always* be sequentialized provided there exists a *compositional proof* of correctness of the concurrent program. More precisely, we show that given a concurrent program C with assertions and a set of *auxiliary* variables A, there is a sequentialization of it, $S_{C,A}$ with assertions, and that C can be shown to compositionally satisfy its assertions by exposing the auxiliary variables A if and only if $S_{C,A}$ satisfies its assertions. The notion of C compositionally satisfying its assertions using auxiliary variables A is defined semantically, and intuitively captures the *rely-guarantee proofs* pioneered by Jones [9]. Rely-guarantee proofs of concurrent programs are very standard, and perhaps the best known compositional verification technique for concurrent programs. In these proofs, *auxiliary variables* can be seen as local states that get exposed in order to build a compositional rely-guarantee proof.

Compositional proofs of programs may not always exist, and since our sequentialization only produces sequential programs that are precise when a compositional proof exists over the fixed auxiliary variables A, proving its sequentialization correct can be seen as a sound but incomplete mechanism for verifying the concurrent program. Note that our sequentialization *does not* require the compositional proof to be given; it is only parameterized by the auxiliary variables A. In fact, if the sequential program is correct, then we show that the concurrent program is always correct. Conversely, if the concurrent program is correct and has a compositional proof using variables A, then we show that the sequential program is guaranteed to be correct as well.

The salient aspect of our sequentialization is that it can be used to prove concurrent programs *entirely* correct, as opposed to checking underapproximations of it. Moreover, though our sequentializations are sound but incomplete, we believe they are useful on most practical applications since concurrent programs often have compositional proofs. Our result also captures the *cost* of se-

quentialization (i.e. the number of variables in the sequentialization) as directly proportional to the number of auxiliary variables that are required to build a compositional proof. Concurrent programs that are "loosely coupled" often require only a small number of auxiliary variables to be exposed, and hence admit efficient sequentializations.

We also describe our experience in applying our sequentialization to prove a suite of concurrent programs entirely correct by using deductive verification of their sequentializations. More precisely, we wrote rely-guarantee proof annotations for some concurrent programs by formulating the rely and guarantee conditions, the loop invariants, and pre- and post-conditions for every function. We then sequentialized the concurrent program and also *transformed* the rely-guarantee proof annotations to corresponding proof annotations on the sequential program. As we show, in this translation, rely and guarantee conditions naturally get transformed to pre- and post-conditions of methods, while loop-invariants and pre- and post-conditions get translated to loop invariants and pre- and post-conditions in the sequential program. Then, using an automatic sequential program verifier BOOGIE, we verified the sequentializations correct. BOOGIE takes our programs with the proof annotations, generates verification conditions, and discharges them using an automatic theorem prover (SMT solver).

The above use of sequentialization for deductive verification is not the best use of our sequentializations, as given rely-guarantee proofs, simpler techniques for statically verifying them are known [6]. However, our sequentializations can be applied even when the rely-guarantee proofs are not known, provided the sequential verification tool is powerful to prove it correct. Indeed, we have also used the sequentialization followed by an automatic predicate-abstraction tool (SLAM [2]) to prove some concurrent programs correct.

In summary, the result presented in this paper shows a surprising connection between compositional proofs and sequentializability. We believe that this constitutes a fundamental theoretical understanding of when concurrent programs are efficiently sequentializable, and offers the first efficient sequentializations that work without underapproximation restrictions, enabling us to verify concurrent program entirely using sequentializations.

Related work: Thread-modular verification [6, 7] is in fact precisely the same as compositional verification á la Jones, but has been adapted to both model-checking [7] and extended static checking [6]. Our result can be hence seen as showing how thread-modular verification of concurrent programs can be reduced to pure sequential verification. There has also been work on using counter-example guided predicate-abstraction and refinement for rely-guarantee reasoning [4], and building rely-guarantee interfaces using *learning* [3, 1].

2 A compositional abstract semantics for programs

We define a *non-standard compositional semantics* for concurrent programs, different from the traditional semantics, in order to capture when a parallel composition of programs can be *argued compositionally to satisfy a specification*. This

semantics is parameterized by a set of auxiliary variables, and is the semantic analog of *compositional rely-guarantee proofs* pioneered by Jones [9].

Let us fix two processes P_1 and P_2, working concurrently, with local variables L_1 and L_2 respectively, and a set of shared variables S (assume L_1, L_2 and S are pairwise disjoint, without loss of generality). For any set of (typed) variables V, let Val_V denote the set of valuations of V to their respective data-domains (data-domains are finite or countably infinite). For any $u \in Val_V$, let $u \downarrow V'$ denote the valuation u restricted to the variables in $V \cap V'$. We extend this notation to sets of valuations, $U \downarrow V'$. Also, for any $u \in Val_V$ and $u' \in Val_{V'}$, where $V \cap V' = \emptyset$, let $u \cup u'$ denote the unique valuation in $Val_{V \cup V'}$ that extends u and u' to $V \cup V'$.

Let $Init \subseteq (Val_{L_1} \times Val_{L_2} \times Val_S)$ be the set of initial global configurations of $P_1||P_2$. Let $\delta_1 \subseteq (Val_{L_1} \times Val_S \times Val_{L_1} \times Val_S)$ and $\delta_2 \subseteq (Val_{L_2} \times Val_S \times Val_{L_2} \times Val_S)$ be the local transition relations of P_1 and P_2, respectively.

The natural (interleaving) semantics of $P_1||P_2$ is, of course, defined by the function $\delta \subseteq (Val_{L_1} \times Val_{L_2} \times Val_S \times Val_{L_1} \times Val_{L_2} \times Val_S)$, where $\delta(l_1, l_2, s, l_1', l_2', s')$ holds iff $\delta_1(l_1, s, l_1', s')$ holds and $l_2' = l_2$, or $\delta_2(l_2, s, l_2', s')$ holds and $l_1' = l_1$. The set of *reachable states* according to this relation, *Reach*, is defined as the set of global states that can be reached from the initial state.

Let us now define the non-standard compositional semantics of $P_1||P_2$. This definition is parameterized by a set of *auxiliary variables* $A \subseteq L_1 \cup L_2$.

Definition 1. *The semantics of the compositional semantics of parallel composition with respect to the set of auxiliary variables A, denoted* $P_1||_A P_2$*, is defined using the* four *sets:*

$$R_1 \subseteq (Val_{L_1} \times Val_S \times Val_{A \cap L_2}),$$
$$R_2 \subseteq (Val_{L_2} \times Val_S \times Val_{A \cap L_1}),$$
$$Guar_1, Guar_2 \subseteq (Val_S \times Val_A \times Val_S \times Val_A),$$

which are defined as the least *sets that satisfy the following conditions:*

a) Initialization:
- R_1 *contains the set* $\{(l_1, s, t) \mid l_1 \cup s \cup t \in Init \downarrow (L_1 \cup A \cup S)\}$.
- R_2 *contains the set* $\{(l_2, s, t) \mid l_2 \cup s \cup t \in Init \downarrow (L_2 \cup A \cup S)\}$.

b) Transitions of P_1**:** *If* $(l_1, s, t) \in R_1$ *and* $\delta_1(l_1, s, l_1', s')$ *holds, then*
- **Local update:** $(l_1', s', t) \in R_1$.
- **Update to guarantee:** $(s, l_1 \downarrow A \cup t, s', l_1' \downarrow A \cup t) \in Guar_1$.

c) Transitions of P_2**:** *If* $(l_2, s, t) \in R_2$ *and* $\delta_2(l_2, s, l_2', s')$ *holds, then*
- **Local update:** $(l_2', s', t) \in R_2$
- **Update to guarantee:** $(s, l_2 \downarrow A \cup t, s', l_2' \downarrow A \cup t) \in Guar_2$.

d) Interference:
- *If* $(l_1, s, t) \in R_1$ *and* $(s, l_1 \downarrow A \cup t, s', t') \in Guar_2$*, then* $(l_1, s', t' \downarrow L_2) \in R_1$.
- *If* $(l_2, s, t) \in R_2$ *and* $(s, l_2 \downarrow A \cup t, s', t') \in Guar_1$*, then* $(l_2, s', t' \downarrow L_1) \in R_2$.

The set of reachable states *according to the* non-standard compositional semantics *with respect to the set of auxiliary variables A is defined as*

$$Reach_A = \{(l_1, s, l_2) \mid (l_1, s, l_2 \downarrow A) \in R_1 \text{ and } (l_2, s, l_1 \downarrow A) \in R_2\}. \qquad \square$$

Intuitively, under the compositional semantics, we track independently the view of P_1 (and P_2) using valuations of its local variables, shared variables, and the subset of the other process's local variables declared to be auxiliary (using the sets R_1 and R_2). Furthermore, we keep the set of *guarantee* transition-relations $Guar_1$ and $Guar_2$ that summarize what transitions P_1 and P_2 can take, but restricted to the auxiliary and shared variables only. The guarantee-relation of P_1 is used to update the view of P_2 (i.e. R_2), and vice versa. The crucial aspect of the definition above is that it *ignores* the correlation between local variables of P_1 and P_2 that are not defined to be auxiliary variables. The computation of $P_1||_A P_2$ hence proceeds mostly locally, with updates using the guarantee relation of the other process (which affects shared and auxiliary variables only), and is combined in the end to get the set of globally reachable configurations.

It is not hard to see that $Reach \subseteq Reach_A$, for any A. Hence, the compositional semantics is an *over-approximation* of the set of reachable states of the program, and proving that a program is safe under the compositional semantics is sufficient to prove that the program is safe. Moreover, when the auxiliary variables include all local variables (including the program counter and local call stack), the compositional semantics coincides with the natural semantics.

The above definitions and rules can be generalized to k processes running in parallel, and we can define the compositional semantics $P_1||_A P_2||_A \ldots ||_A P_n$ where A is subset of local variables of each process.

The rely-guarantee proof method of Jones:

The rely-guarantee method of Jones [9] essentially builds compositional rely-guarantee *proofs* using a similar abstraction. Given *sequential programs* P_1 and P_2, and a pre-condition *pre* and a post-condition *post* for $P_1||P_2$, the rely-guarantee proof technique over a set of auxiliary variables A involves providing a pair of tuples, $(pre_1, post_1, rely_1, guar_1)$ and $(pre_2, post_2, rely_2, guar_2)$, where pre_1, $post_1$, pre_2, and $post_2$ are unary predicates defining subsets of states, and $rely_1$, $guar_1$, $rely_2$, $guar_2$ are binary relations defining transformations of the shared variables and the auxiliary variables A. The meaning of the tuple for P_1 is that, when P_1 is started with a state satisfying pre_1 and in an environment that could change the auxiliary variables and shared variables allowed by $rely_1$, P_1 would make transitions that accord to $guar_1$, and if it terminates, will satisfy $post_1$ at the exit. An analogous meaning holds for P_2. Note that $rely_1, rely_2, guar_1$ and $guar_2$ are defined over shared variables and the auxiliary variables. The programs P_1 and P_2 are proved to satisfy these conditions using a *local* proof by considering each P_i interacting with a general environment satisfying $rely_i$; in particular invariants of P_i needed to establish the Hoare-style proof of P_i should be *invariant* or stable with respect to $rely_i$.

The following proof rule can then be used to prove partial correctness of $P_1||P_2$:

$$\frac{\begin{array}{cc} guar_1 \Rightarrow rely_2, & guar_2 \Rightarrow rely_1, \\ P \models (pre, post_1, rely_1, guar_1), & Q \models (pre, post_2, rely_2, guar_2) \end{array}}{P||Q \models (pre, post_1 \wedge post_2)}$$

The rely-guarantee method works also for *nested* parallellism compositionally; see [9, 16] for details.

It is easy to see that a compositional rely-guarantee proof of $P_1||P_2$, over a set of auxiliary variables A, is really a proof that the compositional semantics of $P_1||_A P_2$ is correct. Note that if $P_1||_A P_2$ is correct, it does not imply a rely-guarantee proof exists, however, as proofs have limitations of the logical syntax used to write the rely and guarantee conditions, and hence do not always exist.

The main result:
We can now state the main result of this paper. We show that, given a parallel composition of sequential programs $P_1||P_2||\ldots||P_n$ with assertions, and a set of auxiliary variables A, we can build a sequential program S with assertions such that S has the following properties:

- At any point, the scope of S contains at most one copy of the local variables of a *single* process P_i, three copies of the auxiliary variables, and at most three copies of the shared variables.
- The compositional semantics of $P_1||_A P_2||_A \ldots||_A P_n$ with respect to the auxiliary variables A satisfies its assertions iff S satisfies its assertions.
- If S satisfies its assertions, then $P_1||P_2||\ldots||P_n$ also satisfies its assertions.

The first remark above says that the sequentialized program has less variables in scope than the naive *product* of the individual processes; the sequentialization intuitively simulates the processes *separately*, keeping track of only an extra copy of auxiliary variables and shared variables. Second, the sequentialization is a *precise* reduction of the verification problem, provided the concurrent program can be proved compositionally (i.e. if the auxiliary variables are sufficient to make the compositional semantics of the program be assertion-failure free). Finally, the sequential program is an over-approximation of the behaviors of the parallel program for *any* set of auxiliary variables, and hence proving it correct proves the parallel program correct.

The above result will be formalized in the sequel (see Theorem 1) for a class of parallel programs that has sequential recursive functions, but with no thread creation or dynamic memory allocation (the result can be extended to dynamic data-structures but will require mechanisms to cache heap-structures and compare them for equality). Our main theorem hence states that any parallel program that is amenable to compositional reasoning can be sequentialized, where the number of new variables added in the sequentialization grows with the number of auxiliary variables required to prove the program correct. We utilize the sequentialization result in one verification context, namely deductive verification, to build a compositional deductive verification tool for concurrent programs using the sequential verifier Boogie.

3 A high level overview of the sequentialization

In this section, we give a brief overview of our sequentialization. For ease of explanation, let us consider a concurrent program consisting of two processes P_1

and P_2. Let A be the set of auxiliary variables and assume that the compositional semantics of the concurrent program is correct with respect to A.

Assume we had functions $G_1(s^*, a^*)$ (and $G_2(s^*, a^*)$) that *somehow* takes a shared and auxiliary state (s^*, a^*) and non-deterministically returns all states (s, a) such that $(s^*, a^*, s, a) \in \mathit{Guar}_1$ (respectively Guar_2), where Guar_1 and Guar_2 are as in Definition 1. Then we could write a function that computes the states reachable by P_1 according to the compositional semantics (i.e. R_1 in Definition 1) using the following code:

```
while(*) {
  if (*) then
    <<simulate a transition of P1>>
  else
    (s,a) := G2(s,a);
  fi
}
return (s,a);
```

In other words, we could write a sequential program that returns precisely the states in R_1, by interleaving simulations of P_1 with calls to G_2 to compute interference according to Guar_2 (see Definition 1). We can similarly implement the sequential code that explores R_2 using calls to G_1.

Note that on two *successive* calls to $G_2()$, there is no preservation of the local states of P_2, *except its variables declared to be auxiliary*. However, we do *not* have to preserve the exact local state of P_2 as we are not simulating the natural semantics of the program, but only its compositional semantics with respect to auxiliary variables A. This is the crux of the argument as to why we can sequentially compute R_1 without simultaneously tracking all the local variables of P_2.

Now, turning to the function G_2 (and G_1), consider Definition 1 again, and notice that, given (s^*, a^*), in order to compute (s, a) such that $(s^*, a^*, s, a) \in \mathit{Guar}_2$, we must essentially be able to find a local state l_2 such that $(l_2, s^*, a^* \downarrow L_1) \in R_2$ where $l_2 \downarrow A = a^* \downarrow L_2$, and then we can take its transitive closure with respect to δ_2. We can hence write G_2 using the following sequential code:

```
G2(s*,a*) {
   <<initialize variables of P2>>
   while(*) {
       if (*) then
          <<simulate a transition of P2>>
       else
          (s,a) := G1(s,a);
       fi
   }
   assume (local and shared state is consistent with s*,a*);
   while(*) {
        <<simulate a transition of P2>>
   }
   return (s,a);
}
```

Intuitively, G_2 starts with the *initial* state of P_2 and sets about recomputing a state (l_2, s_2) that is compatible with its given input (s^*, a^*) (i.e. with $s_2 = s^*$ and $l_2 \downarrow A = a^* \downarrow L_2$). It does this by essentially running the code for R_2 (i.e. by simulating P_2 and calling G_1). Once it has found such a state, it simulates P_2 for a while longer, and returns the resulting state.

We hence get four procedures that compute R_1, R_2, $Guar_1$ and $Guar_2$, respectively, with mutually recursive calls between the functions computing $Guar_1$ and $Guar_2$. The correctness of the sequential programs follow readily from Definition 1, as it is a direct encoding of that computation. Our sequentialization transformation essentially creates these functions G_1 and G_2. However, since our program cannot have statements like "simulate a transition of P2", we perform a syntactic transformation of the concurrent code into a sequential code, where control code is inserted between statements of the concurrent program in order to define the functions G_1 and G_2. This complication combined with the handling of recursive functions makes the translation quite involved; however, the above explanation captures the crux of the construction.

4 Sequential and concurrent programs

Our language for concurrent programs consists of a parallel composition of recursive sequential programs. Variables in our programs are defined over integer and Boolean domains. The syntax of programs is defined by the following grammar:

$$
\begin{array}{lll}
\langle \textit{conc-pgm} \rangle & ::= & \langle \textit{decl} \rangle^* \langle \textit{pgm-list} \rangle \\
\langle \textit{pgm-list} \rangle & ::= & \langle \textit{pgm-list} \rangle \;||\; \langle \textit{pgm-list} \rangle \mid \langle \textit{pgm} \rangle \\
\langle \textit{pgm} \rangle & ::= & \langle \textit{decl} \rangle^* \langle \textit{proc} \rangle^* \\
\langle \textit{proc} \rangle & ::= & f(\overline{x})\ \texttt{begin}\ \langle \textit{decl} \rangle^* \langle \textit{stmt} \rangle\ \texttt{end} \\
\langle \textit{stmt} \rangle & ::= & \langle \textit{stmt} \rangle; \langle \textit{stmt} \rangle \mid \texttt{skip} \mid \overline{x} := \textit{expr}(\overline{x}) \mid \overline{x} := f(\overline{y}) \mid \\
 & & f(\overline{x}) \mid \texttt{return}\ \overline{x} \mid \texttt{assume}\ \textit{b-expr} \mid \texttt{assert}\ \textit{b-expr} \mid \\
 & & \texttt{if}\ \textit{b-expr}\ \texttt{then}\ \langle \textit{stmt} \rangle\ \texttt{else}\ \langle \textit{stmt} \rangle\ \texttt{fi} \mid \\
 & & \texttt{while}\ \textit{b-expr}\ \texttt{do}\ \langle \textit{stmt} \rangle\ \texttt{od} \mid \texttt{atomic}\ \{\langle \textit{stmt} \rangle\} \\
\langle \textit{decl} \rangle & ::= & \texttt{int}\ \langle \textit{var-list} \rangle; \mid \texttt{bool}\ \langle \textit{var-list} \rangle; \\
\langle \textit{var-list} \rangle & ::= & \langle \textit{var-list} \rangle, \langle \textit{var-list} \rangle \mid \langle \textit{literal} \rangle
\end{array}
$$

A concurrent program consists of k sequential program components $P_1 ... P_k$ (for some k) communicating with each other through shared variables $\mathcal{S}$. These shared variables are declared in the beginning of the concurrent program (we assume integer variables are initialized to 0 and Boolean variables to *false*). Each sequential component consists of a procedure called *main* and a list of other procedures. The control flow for all sequential components P_i starts in the corresponding *main* procedure, which we call $main_i$. The *main* for all sequential components has *zero* arguments and no return value.

Each procedure is a declaration of local variables followed by a sequence of statements, where statements can be simultaneous assignments, function calls

(call-by-value) that take in multiple parameters and return multiple values, conditionals, while loops, assumes, asserts, atomic, and return statements. In the above syntax, $\overline{x}$ represents a vector of variables. We allow non-determinism in our programs; boolean constants are *true*, *false* and $*$, where $*$ evaluates non-deterministically to *true* or *false*.

The safety specifications for both concurrent and sequential programs are expressed in our language as assert statements. The semantics of an assume statement is slightly different. If the value of the boolean expression (*b-expr*) evaluates to *true*, then the assume behaves like a skip. Otherwise, if the boolean expression evaluates to *false*, the program silently terminates. Synchronization and atomicity are achieved by the *atomic* construct. All the statements enclosed in the atomic block are executed without any interference by the other processes. Locks can be simulated in our syntax by modeling a lock l as an integer variable `l` and by modeling P_i acquiring l using the code:

```
atomic { assume(l=0); l := i;}
```

and modeling the release with the code:

```
atomic { if (l=i) then l := 0;}
```

We assume programs do not have nested atomic blocks.

The syntax of sequential programs is the same as the syntax of concurrent programs except that we disallow the parallel composition operator ($||$) and the *atomic* construct.

5 The Sequentialization

In this section, we describe our sequentialization for concurrent programs and argue its correctness.

Let us fix a concurrent program with shared variables S and auxiliary variables A; we assume auxiliary variables are *global* in each thread P_i. Let the concurrent program be composed of k sequential components.

The sequential program corresponding to the concurrent program will have a new function *main*, and additionally, as explained in Section 3, will have a procedure G_i for each sequential component P_i of the concurrent program that semantically captures the guarantee $Guar_i$ of P_i. The procedure G_i takes a shared state ($\overline{s^*}$) and auxiliary state ($\overline{a^*}$) as input and returns ($\overline{s}, \overline{a}$) such that ($\overline{s^*}, \overline{a^*}, \overline{s}, \overline{a}$) $\in$ $Guar_i$. Finally, each $G_i()$ is formed using procedures that are obtained by transforming the process P_i (using the function $\tau_i[]$ shown below that essentially *inserts* the interference code $\mathcal{I}_i$ shown in Figure 1 between the statements of P_i).

The shared variables and auxiliary variables are modeled as global variables in the sequential program. Furthermore, we have an extra copy of the shared and auxiliary variables ($\overline{s^*}$ and $\overline{a^*}$) that are used to pass shared and auxiliary states between the processes $G_i()$. We also have a copy of shared and auxiliary variables ($\overline{s'}$ and $\overline{a'}$) that are declared to be *local* in each procedure to store a shared and auxiliary state and restore it after a call to a function $G_j()$ to compute interference. Besides these, the sequential program also uses global

Boolean variables z and *term*; intuitively, z is used to keep track of when the shared and auxiliary state $\overline{s^*}$ and $\overline{a^*}$ has been reached and *term* (for *terminate*) is used to signal that $G_i()$ has finished computing and wants to return the value.

– Global variable declarations are:

```
// insert declaration for s̄, ā, s̄*, ā* as global variables
decl bool term, z;
```

– The function main() is defined as: `main() begin G_1() end`

– Each function $G_i()$ is defined as below:

```
G_i() begin
   z := false;  term := false;
   s̄* := s̄;  ā* := ā;
   // insert code to initialize s̄, ā
   main_i();
   assume (term = true);
   return
end
```

– The function τ_i that transforms the program for P_i is defined as:

- τ_i[`f(`$\overline{\mathtt{x}}$`) begin` *decl stmt* `end`] =
  `f(`$\overline{\mathtt{x}}$`) begin` *decl*
  `// insert declaration of` $\overline{\mathtt{s}'}, \overline{\mathtt{a}'}$ `as local variables.`
  $\tau_i[stmt]$
  `end`
- $\tau_i[\mathcal{S}_1;\ \mathcal{S}_2] = \tau_i[\mathcal{S}_1];\ \tau_i[\mathcal{S}_2]$
- $\tau_i[\mathcal{S}] = \mathcal{I}_i;\ \mathcal{S}$ where $\mathcal{S}$ is an assignment, skip, assume, assert, a function call or a return statement.
- τ_i[`while` *b-expr* `do` $\mathcal{S}$ `od`] = $\mathcal{I}_i$; `while` *b-expr* `do` $\tau_i[\mathcal{S}]$; $\mathcal{I}_i$ `od`
- τ_i[`if` *b-expr* `then` $\mathcal{S}_1$ `else` $\mathcal{S}_2$ `fi`] =
  $\mathcal{I}_i$; `if` *b-expr* `then` $\tau_i[\mathcal{S}_1]$ `else` $\tau_i[\mathcal{S}_2]$ `fi`
- τ_i[`atomic` $\{\mathcal{S}\}$`]` = $\mathcal{I}_i;\ \mathcal{S}$

The procedure *main* in the sequential program simply calls the method G_1.

The procedure G_i is obtained from the corresponding program component P_i by a simple transformation. At a high level, this procedure first copies the incoming shared and auxiliary state into the variables $\overline{s^*}$ and $\overline{a^*}$. It then computes a local state of P_i which is consistent with the state $(\overline{s^*}, \overline{a^*})$ (at which point z turns to *true*), and then non-deterministically simulates the transitions of P_i from this local state, to return a reachable shared state $\overline{s}$ and auxiliary state $\overline{a}$. Every time G_i is called, it starts from its initial state, and simulates P_i, interleaving it with the control code $\mathcal{I}_i$ given in Figure 1.

The interference code $\mathcal{I}_i$ (Figure 1) keeps track of whether the incoming state $(\overline{s^*}, \overline{a^*})$ has been reached through a boolean variable z which is initialized to *false*. If z is *false* (i.e. the state $(\overline{s^*}, \overline{a^*})$ has not been reached), then before

any transition of P_i, the control code can non-determinsitically choose to invoke its environment (in doing so, in order to preserve its input $\overline{s^*}, \overline{a^*}$, it stores them in a local state and restores them after the call returns and restores its variables z and *term* to *false*).

When the state $(\overline{s^*}, \overline{a^*})$ is reached, z can be non-deterministically set to *true*, from which point no interference code G_j can be called, and only local computation proceeds, till non-deterministically the program decides to terminate by setting *term* to *true*. Once *term* is *true*, the code pops the control-stack all the way back to reach the function G_i which then returns to its caller, returning the new state in $(\overline{s}, \overline{a})$. Note that the state $z = false$, $term = false$ corresponds to the first while loop in the code for the guarantee G_2 in Section 3. Similarly, setting *term* to *true* corresponds to the termination of the second while loop. We conclude this section by stating our main theorem:

```
if(term = true) then return fi
if(!z & *) then
  while(*) do
    // call G_1
    if(*) then
          s'_bar := s*_bar;  a'_bar := a*_bar;
          G_1();
          z := false;
          term := false;
          s*_bar := s'_bar;  a*_bar := a'_bar
    fi
    Similarly call G_2 ... G_k except G_i
  od
fi
if(!z & s_bar = s*_bar & a_bar = a*_bar & *) then
  z := true fi
if(z & *) then
  term := true; return
fi
```

Fig. 1. The interference control code $\mathcal{I}_i$

Theorem 1. *Let C be a concurrent program with auxiliary variables A (assumed global), and let $S_{C,A}$ be its sequentialization with respect to A. Then the compositional semantics of C with respect to the auxiliary variables A has no reachable state violating any of its assertions iff $S_{C,A}$ violates none of its assertions.* □

Illustration of a sequentialization:
Figure 2 shows a concurrent program consisting of two threads, say P_1 and P_2. The program consists of a shared variable x whose initial value is *zero*. Both the threads *atomically* increment the value of x. Let $A = \{pc1, pc2\}$ be the auxiliary variables capturing the control position in the respective processes and let the initial value of these variables also be *zero*. In general, new auxiliary variables may be needed for performing compositional proofs; these new variables are *written to* but never read from in the program, and hence do not affect the semantics of the original program; see [9, 15].

It can be easily seen that the compositional semantics of this program with respect to the auxiliary variables A is correct. Figure 3 shows the sequential program obtained from the sequentialization of this concurrent program with respect to these auxiliary variables. Our result allows us to verify the concurrent program in Figure 2 by verifying the correctness of its sequentialization with respect to the auxiliary variables A, shown in Figure 3.

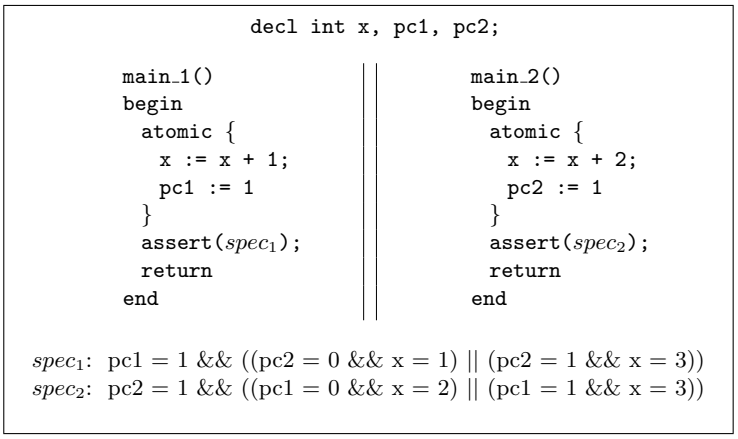

Fig. 2. An example program

6 Experience

The sequentialization used in this paper can be used to verify a concurrent program using *any* sequential verification tool. This includes tools based on abstract interpretation and predicate abestraction, those based on bounded model-checking, as well as those based on deductive-verification based extended static checking.

Deductive verification: We used the sequentialization for proving concurrent programs using deductive verification. Given a concurrent program and its Jones-style rely-guarantee proof annotations (pre, post, rely, guar, and loop-invariants), we sequentialized it with respect to the auxiliary variables and syntactically transformed the user-provided proof annotations to obtain the proof annotations (pre-conditions and post-conditions of methods and loop invariants) of the sequential program. In general, the pre-condition of method G_i asserts that "$term = false$" and its post-condition asserts that the guarantee $guar_i$ is true across the function (if *term* is *true*). Furthermore, the pre-conditions and post-conditions of every function in P_i gets translated to pre-conditions and post-conditions in its sequentialization with the extra condition that $guar_i$ is *true* when *term* is equal to *true*.

These annotated sequential programs were fed to the sequential verifier BOOGIE that generates verification conditions that are in turn solved by an SMT solver (Z3 in this case). If the sequential program is proved correct, it proves the correctness of the original concurrent program. Note that though similar static extended checking techniques are known for the rely-guarantee method [6], our technique allows one to use just a sequential verifier like BOOGIE to prove the program correct, and requires no other decision problems to be solved (the checking in [6], for example, requires a separate call to a theorem prover to check guarantees are reflexive and transitive, etc.)

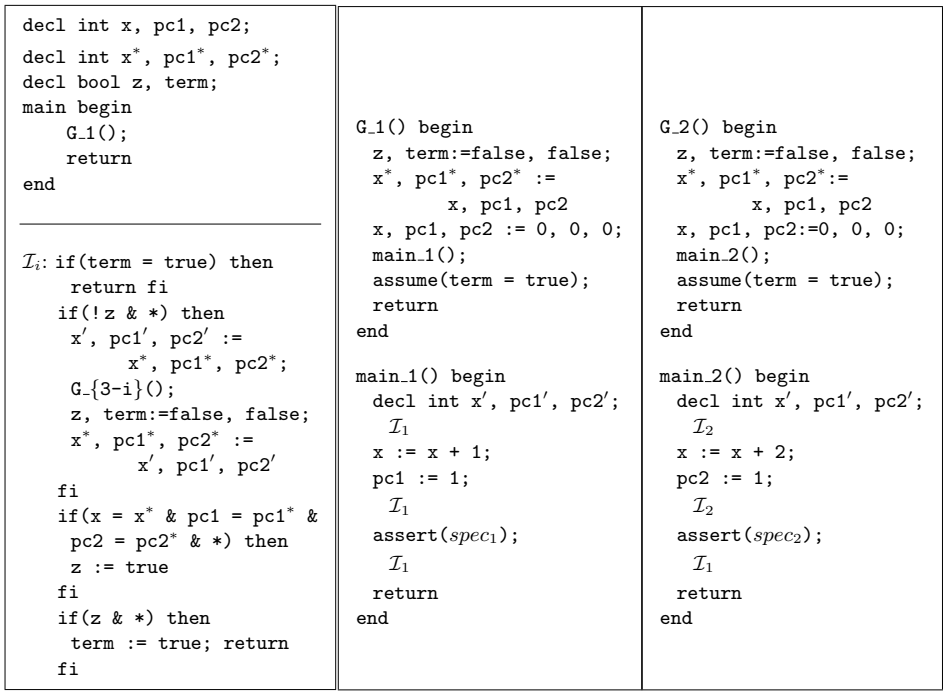

Fig. 3. The sequential program obtained from the concurrent program in Figure 2.

We used this technique to prove correct the following set of concurrent programs: X++ (Figure 2), Lock [7], Peterson's mutual exclusion algorithm, the Bakery protocol, ArrayIndexSearch [9], GCD [5], and a simplified version of a Windows NT Bluetooth driver. Lock is a simple example program consisting of two threads that modify a shared variable after acquiring a lock; the safety condition in the example asserts that these modifications cannot occur concurrently. The program ArrayIndexSearch finds the *least index* of an array such that the value at that index satisfies a given predicate, and consists of two threads, one that searches odd indices and the other that searches even indices, and communicate on a shared variable *index* that is always kept updated to the current least index value. GCD is a concurrent version of Euclid's algorithm for computing the greatest common divisor of any two numbers; here the two concurrent threads update the pair of integers.

The Windows NT bluetooth driver is a *parameterized program* (i.e. has an unbounded number of threads). It consists of two types of threads: there is one stopper-thread and an unbounded number of adder-threads. A stopper calls a procedure to halt the driver, while an adder calls a procedure to perform I/O in the driver. The I/O is successfully handled if the driver is not stopped while it executes. The program, though small, has an intricate global invariant that requires a shared variable to reflect the number of active adder threads.

	Concurrent pgm			Sequential pgm		
Programs	#Threads	#LOC	#Lines of annotations	#LOC	#Lines of annotations	Time
X++	2	38	5	113	6	8s
Lock	2	50	9	184	10	122s
Peterson	2	52	35	232	36	145s
Bakery	2	55	8	147	13	18s
ArrayIndexSearch	2	74	17	222	21	126s
GCD	2	78	23	279	29	869s
Bluetooth	unbdd	69	20	276	55	107s

Table 1. Experimental Results. Evaluated on Intel dual-core 1.6GHz, 1Gb RAM.

Table 1 gives the experimental results[1]. For each program, we report the number of threads in the concurrent program, the number of lines of code in the concurrent program and its sequentialization, the number of lines of annotations in both the concurrent program (which includes rely/guarantee annotations and loop invariants) and its sequentialization, and the time taken by BOOGIE to verify the sequentialized program.

BOOGIE was able to verify the correctness of all our programs. All these programs except the Windows NT bluetooth driver consist of two threads and are sequentialized as detailed in Section 5. The Bluetooth driver is an example of a parameterized program running any number of instances of the adder threads. In our sequentialization, we model the environment consisting of all the adder threads together with a single procedure. If we keep track of the number of adders at a particular program location (counter abstraction [14]) and expose these auxiliary variables, it turns out that the device driver can be proved correct under compositional semantics. We used this rely-guarantee proof, sequentialized the program, and used BOOGIE to prove the Bluetooth driver correct in its full generality.

Predicate abstraction: We have also used our sequentialization followed by the predicate-abstraction tool SLAM [2] to prove programs automatically correct. In this case, we need no annotations and just the set of auxiliary variables. We were able to automatically prove the correctness of the programs X++, Lock, Peterson and the Bakery protocol, in negligible time.

7 Future directions

One drawback of our sequentialization is that it creates recursive programs, even when the concurrent program has no recursion. It is hence natural to ask whether there is a sequentialization that does not introduce recursion. We know indeed

[1] Experiments available at http://www.cs.uiuc.edu/∼garg11/tacas11

of such a sequentialization, where the process G_i does not start from the *initial* state to match the state given to it, but rather "jumps" directly to a state consistent with the state given to it. This, of course, does not capture compositional semantic reasoning precisely, and seems to be an over-approximation of it. Evaluating the effectiveness of this translation is an interesting future direction.

Finally, since our sequentialization captures the concurrent program without under-approximations, utilizing the sequentialization followed by techniques such as static analysis, predicate-abstraction, and even bounded model-checking, would be interesting directions to pursue.

Acknowledgements: This work is partially supported by NSF grants #0747041 and #1018182.

References

1. R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *CAV*, volume 3576 of *LNCS*, pages 548–562. Springer, 2005.
2. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3. ACM, 2002.
3. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, volume 2619 of *LNCS*, pages 331–346. Springer, 2003.
4. A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. In *CAV*, volume 4590 of *LNCS*, pages 55–67. Springer, 2007.
5. X. Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327. ACM, 2009.
6. C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *ESOP*, volume 2305 of *LNCS*, pages 262–277. Springer, 2002.
7. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, volume 2648 of *LNCS*, pages 213–224. Springer, 2003.
8. N. Ghafari, A. J. Hu, and Z. Rakamarić. Context-bounded translations for concurrent software: An empirical evaluation. In *SPIN*, volume 6349 of *LNCS*, pages 227–244. Springer, 2010.
9. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.
10. S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV*, volume 5643 of *LNCS*, pages 477–492. Springer, 2009.
11. S. La Torre, P. Madhusudan, and G. Parlato. Sequentializing parameterized programs. Available at http://www.cs.uiuc.edu/∼madhu/seqparam.pdf, 2010.
12. S. K. Lahiri, S. Qadeer, and Z. Rakamarić. Static and precise detection of concurrency errors in systems code using SMT solvers. In *CAV*, volume 5643 of *LNCS*, pages 509–524. Springer, 2009.
13. A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *CAV*, volume 5123 of *LNCS*, pages 37–51. Springer, 2008.
14. B. D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs. *Acta Inf.*, 21:125–169, 1984.
15. S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Inf.*, 6:319–340, 1976.
16. Q. Xu, W. P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Asp. Comput.*, 9(2):149–174, 1997.