Verifying Concurrent Programs via Bounded Context-Switching and Induction [*]

Prathmesh Prabhu and Thomas Reps[*]
Univ. of Wisconsin; Madison, WI USA
[*]GrammaTech, Inc.; Ithaca, NY, USA
{pprabhu,reps}@cs.wisc.edu

Akash Lal
Microsoft Research India
Bangalore, India
akashl@microsoft.com

Nicholas Kidd
Google
Madison, WI USA
nkidd@google.com

Abstract

This paper presents a new approach to the problem of verifying safety properties of concurrent programs with shared memory and interleaving semantics. Our method builds on and extends context-bounded analysis (CBA), in which thread interleavings are considered only up to K context switches. In a K-induction argument, the base case establishes that the property holds for the first K steps (first K context switches in our case); the inductive case establishes that if the property held for the previous K steps (context switches), then it will hold after one more step (context switch). Our approach uses CBA directly to handle the base case, and uses CBA as a subroutine when discharging the inductive case.

The account sketched out above over-simplifies; there are actually several impediments to combining CBA and K-induction. The paper identifies these challenges and introduces three techniques that, when used together, side-step the difficulties.

1. Introduction

Analysis of concurrent programs has been a topic of great interest in recent research. In the general case, analysis of both concurrent and sequential programs is undecidable; however, even with simplified modeling frameworks for which the sequential version of a problem is decidable, the concurrent version of the problem is either much more expensive, or undecidable.

Because verification is so challenging, another approach has been to devise tools that explore only a portion of a concurrent program's state space, as a means for detecting bugs. For instance, context-bounded analysis (CBA) [4, 13, 17] analyzes all behaviors of a concurrent program for up to K context switches, but ignores behaviors that involve more than K context switches. While CBA cannot prove the absence of bugs, empirical results have shown that it is able to capture many of the interesting behaviors of a program [13, 16]. If CBA cannot reveal a bug within a few context switches, this is a strong indication that the program is correct.

This paper develops a technique for verifying safety properties of concurrent programs (with shared memory and interleaving semantics) by combining methods adapted from CBA with a rule of induction that generalizes K-induction [1, 6, 8, 9, 20]. In some sense, it can be thought of as a technique to generalize the information learned from a CBA run to construct a proof of correctness.

[*] Supported, in part, by NSF under grants CCF-{0810053, 0904371}, by ONR under grants N00014-{09-1-0510, 10-M-0251}, by ARL under grant W911NF-09-1-0413, and by AFRL under grants FA9550-09-1-0279 and FA8650-10-C-7088. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

Applying K-induction to concurrent programs comes with its own set of challenges, which force some modifications to be made to the basic proof technique. For instance, one needs to find a way to have the inductive hypothesis consider only a restricted set of start states for the inductive case. The paper discusses the challenges and develops an appropriate version of K-induction that is suitable for use on concurrent programs, which we call *K-induction with amplification*.

Given a concurrent program P and a safety property A, K-induction with amplification breaks down the task of proving that P satisfies A into (roughly) the following two proof obligations:

Base case: Prove that the required property holds after all executions that start at the program's initial state and perform up to K context switches.

Inductive case: Prove that, starting in an arbitrary state, the property either (i) fails to hold after *some* execution that performs up to K context switches, or (ii) holds in *all* executions that perform up to $K + 1$ context switches.

(K-induction with amplification also has a further ingredient, which allows the inductive hypothesis to consider a more restricted set of start states for the inductive case. See §2 and §4.)

One of the chief motivations for combining K-induction with CBA is that although the proof obligations refer to a bounded number of context switches, a context-bounded run still contains execution sequences of unbounded length between context switches. This observation suggests that K-induction plus CBA should be more powerful than a K-induction technique that is based on induction over single execution steps.

The verification of safety properties for recursive concurrent Boolean programs is known to be undecidable, and thus a K-inductive proof using K-induction with amplification may not exist for some property that indeed holds for P. In keeping with classical terminology, we say that a property that can be proved using K-induction with amplification for some bound K is "K-inductive". The paper presents a semi-decision procedure based on K-induction with amplification for recursive Boolean programs, and identifies a class of K-inductive programs for which the algorithm terminates.

Contributions. The contributions of the work can be summarized as follows:

- We identify the challenges involved in employing K-induction to prove safety properties of concurrent programs.
- We propose a trio of techniques that, when used together, address the challenges. The workhorse is CBA (and some necessary manipulations to allow CBA to apply in the inductive case of a K-induction proof). However, CBA alone is insufficient. In particular, we employ two further techniques that, in effect,

```
[1] bool x = true;
[2] error : goto error;
[3]
[4] void rec_fun() {
[5]    if(*)
[6]       rec_fun();
[7]    else
[8]       return;
[9] }

[10] void thread1() {
[11]    rec_fun();
[12]    if( !x )
[13]       goto error;
[14] }

[15] void thread2() {
[16]    rec_fun();
[17] }
```

Figure 1. A program with two threads. Lines [1]–[9] are shared declarations and functions. Function thread1() is the entry point of the first thread and thread2() of the second thread. rec_fun is a recursive function that nondeterministically calls itself or returns. The label error (line [2]) is not reachable: x is initialized to *true* and never changed; hence, the true branch of the test in line [12] can never be taken.

prune states from being considered as potential start states for the inductive case: (i) Claessen's method of "Improved Induction" [6] (except that induction is over context switches rather than program steps), and (ii) abstract interpretation to remove from consideration some of the states that are not reachable from the program's initial state.

We call this collection of techniques *K-induction with amplification*. As will be discussed in §2, the advantages of K-induction with amplification stem from three effects:

- it summarizes paths through execution contexts so that arbitrarily long sequences of program actions count as just 1 against the bound of K
- instead of considering all states during the inductive step, it prunes states whose shortest execution-context path to error is smaller than the current window
- some of the states that are unreachable in any execution from the beginning of the program are pruned.

- We report on an implementation of the method using the model checker Moped [19].

Related Work. K-induction [1, 6, 20] has been studied in the hardware model-checking community for analysis of circuits, as well as in the software-verification community for analysis of sequential programs [9].

The first work to use K-induction to analyze concurrent software was by de Moura et al. [8]. They combined K-induction with bounded model checking (BMC) [2]. BMC is a state-space-exploration method that is based on under-approximating a program's semantics (i.e., it may fail to explore some behaviors of the program). In contrast, our work combines K-induction with context-bounded analysis (CBA) [13, 17], which is an alternative to BMC for under-approximating a program's semantics. Both our work and that of de Moura et al. share the goal of using K-induction to augment a core under-approximating method to make it possible to verify properties. At the technical level, the two methods are quite different; a more complete comparison is given in §7.

Organization. §2 uses a small example to illustrate some of the challenges that arise, as well as the main elements of our solution. §3 formalizes K-induction with amplification. §4 presents a program-sequentialization technique that can be used in a K-induction proof. §5 discusses limitations of the approach. §6 presents experimental results. §7 discusses related work.

2. Overview

This section uses the program shown in Fig. 1 to illustrate the challenges that arise when one attempts to apply K-induction to verify safety properties of concurrent programs, as well as the principles that we use to overcome them. The program is written in a C-like notation. Suppose that the goal is to prove the safety property that the program label *error* (line [2]) is unreachable in any execution of the program. For this example, one can argue that *error* is unreachable because x is initialized to *true* and never changed; consequently, the true branch of the test in line [12] can never be taken. However, our goal is to develop a general proof technique, and we would hope that the technique would cover such a simple example, as well.

In Fig. 1, the space of possible executions for each thread when run in isolation is infinite, due to recursion in *rec_fun*. Moreover, the space of possible interleaved executions of the concurrent program is not only infinite, there are now additional behaviors to consider because there can be an unbounded number of context switches in a given run, which are allowed to occur at any execution state.

The need to address *a priori* unbounded behaviors suggests finding a way to use induction. When using induction to verify sequential programs, the challenge is to identify invariants that are inductive over the program and that imply the property of interest. In a K-induction proof, two "windows" of K steps are considered: the base case considers a prefix of up to K steps; the inductive case assumes that the property of interest is true for the previous K steps, and attempts to establish the property for one more step [1, 6, 8, 9, 20]. When a K-induction proof fails, the need to explicitly strengthen the current invariant can sometimes be avoided merely by increasing K. That is, with K-induction, the need to synthesize stronger invariants for the inductive step is alleviated to some extent by the window of K steps: one way to strengthen the invariant is merely to increase the size of the window. Of course, the two approaches—explicit invariant strengthening and increasing K—are independent, and can be used together.

However, several technical challenges arise when trying to use K-induction to verify a concurrent program. In particular, the possibility that a context switch can occur at each execution state makes the use of K-induction challenging in this domain.

CHALLENGE 1 (Effect of context switches on K-induction proofs). *Because a context switch can occur at each execution state, it is not obvious how to push through a K-induction argument. In particular, multiple context switches can occur at an execution state without the program making forward progress—i.e., there can be unbounded stuttering [15].*

Even in the absence of stuttering, the classical definition of K-induction is hampered by the fact that for concurrent programs the least value of K for which a proof is possible can be the length of a very long trace of program steps.

To develop a feasible K-induction technique for verifying concurrent programs, we found that it was necessary to make three adjustments to the K-induction proof rule. These adjustments are based on the two principles introduced below.

PRINCIPLE 1 (Sequences of steps). *Treat sequences of program steps as a single group that collectively count as just* 1 *against the bound of K.*

The main result in §3 shows that techniques adapted from CBA [13, 17] can be combined with K-induction to address Principle 1. An *execution context* is a sequence of program steps executed by a single thread between consecutive context switches. Because CBA focuses on execution contexts, rather than the individual program steps *per se*, it addresses the second of the two issues raised in Challenge 1—namely, the need to reason about long traces of program steps. Because CBA does not impose any bound on the length of an execution context, a summary transformation identified for a single execution context can describe the effect of a long trace of program steps that would otherwise involve a large value of K. For example, a $K = 1$ proof can be found for the case of sequential Boolean programs. This is expected because sequential reachability for Boolean programs is decidable. The fact that the bound K has changed from steps to context switches highlights how the use of K-induction in this paper focuses on a parameter related to concurrency, and summarizes arbitrarily long sequential execution traces.

When K-induction is combined with CBA, the proof obligations break down into a base case involving K context switches and an inductive case involving K (and $K+1$) context switches for some chosen value of K.[1]

EXAMPLE 2.1. *Let us choose K to be 3 for the program shown in Fig. 1. The proof obligations are then to show that*

<u>Base case</u>: *error is not reached in any execution that starts with the program in its initial state and uses up to 3 context switches.*

<u>Inductive case</u>: *starting from an arbitrary program state, if a run does not reach error using up to 3 context switches, then it can not reach error by continuing the run and allowing one additional context switch.*

The intuition behind why one can hope to prove safety properties of concurrent programs via inductive proofs with low values of K—where K is the number context switches, not program steps—is based on the folklore that most bugs in concurrent programs can be found within a few context switches.

There are a number of other reasons why induction over the number of context switches should be beneficial.

- It enables us to reduce the problem to one of reachability in a sequential execution of individual threads, along the lines of the sequentialization transformation used for CBA [13]. In particular, CBA straight out of the box handles the base case.
- A sequentialization reduction can be implemented as a source-to-source transformation, which opens up the possibility of applying essentially any sequential model checker to the problem of verifying concurrent programs.
- In some cases, it enables us to specify a class of programs for which the problem is decidable (because the underlying sequential reachability problem is decidable).

Let us now try to prove the obligations from Ex. 2.1. As we will see, some challenges remain for the inductive case because K-induction over the number of context switches alone is unsatisfactory. It is necessary to introduce some other ingredients to change the problem into a form in which CBA can be applied to the inductive case. However, to understand what needs to be amended, it is instructive to understand what goes wrong with K-induction over the number of context switches alone, and in particular, why it fails to get a handle on long traces of program steps.

EXAMPLE 2.2. *The first proof obligation requires us to prove that error is unreachable within a bounded number of context switches. We can use the sequentialization-based technique developed in [13] to discharge the proof obligation.*

The second proof obligation is more challenging, and is the subject of most of the discussion in the remainder of this section. There are several difficulties:

- *Because the proof obligation for the inductive case says, "starting from an arbitrary program state, . . ." it is necessary to consider execution states in which thread 1 starts at line [13].*
- *As mentioned in Challenge 1, a thread can make a context switch without making any progress, i.e., without executing any program statements. This phenomenon is called* stuttering *[15].*

Consequently, the following execution schedule, where the respective threads have the indicated program-counter values (and the values of the respective stacks are irrelevant) is a valid schedule with 3 context switches:

$$(t2, [15]) \rightsquigarrow (t1, [13]) \rightsquigarrow (t2, [15]) \rightsquigarrow (t1, [13]).$$

Indeed, it is now possible for thread 1 to jump to error without performing a context switch, and so the inductive case fails.

For the induction proof to go through, the least we would need to hope for is to add an artificial assumption about deterministic progress for each thread in every context switch. However, this proposal suggests a second reason why K-induction over the number of context switches runs into difficulties: even in the absence of stuttering, in the worst case, each thread can yield control after executing just a single program step.

EXAMPLE 2.3. *Assume that thread 1 is executing in rec_fun() and that its stack holds a pending activation of procedure thread1$()$ from a call to rec_fun() at line [11]. The following execution schedule has 3 execution contexts (indicated by $\mapsto$), 3 context switches ($\rightsquigarrow$), and no stuttering:*

$$\begin{aligned}(t2, [15]) \mapsto (t2, [16]) \rightsquigarrow (t1, [8]) \mapsto (t1, [13])\\ \rightsquigarrow (t2, [16]) \mapsto (t2, [4]) \rightsquigarrow (t1, [13]).\end{aligned}$$

Once again, thread 1 can now jump to error in one program step (without making an additional context switch), and the inductive case fails.

What is important to note about Ex. 2.3 is that although the induction principle was worded in terms of the number of *context switches*, the specific context switches that occurred in the run effectively caused the K elements of the window to degenerate to individual *program steps*. In other words, for an improved rule to have a chance of using induction over context switches to its advantage, it must address the following challenge—in particular, by following the second of our two principles (see below):

CHALLENGE 2 (Reduction to individual program steps). *An induction technique for verifying concurrent programs should guard against the inductive case degenerating to induction over individual program steps. The induction technique must also be able to handle stuttering caused by repeated context switches that fail to make forward progress.*

PRINCIPLE 2 (Prune start states from the inductive case). *Eliminate as many states as possible from being considered as potential start states for the inductive case.*

We employ two techniques that fall under the rubric of Principle 2:

1. We adopt Claessen's method of "Improved Induction" [6, Defn. 2.3], except that, following Principle 1, the induction is over context switches rather than program steps. Thus, given a concurrent program P and a safety property A, our modified rule of induction breaks down the task of proving that P satisfies A into the following two proof obligations:

   <u>Base case</u>: Prove that the required property holds after all executions that start at the program's initial state and perform up to K context switches.

   <u>Inductive case</u>: Prove that, starting in an arbitrary state, the property either (i) fails to hold after *some* execution that performs up to K context switches, or (ii) holds in *all* executions that perform up to $K+1$ context switches.

   In effect, this revised rule of induction eliminates many states from being considered as potential start states for the inductive case: instead of considering all states during the inductive step (as with K-induction), "Improved Induction" prunes

[1] As will be explained in §3, induction will actually be based on groups of program steps that consist of *multiple* execution contexts, called *epochs*. However, for the purposes of this section, it is sufficient to think of the induction as being over context switches.

states whose shortest path to error is smaller than the current window. In particular, the formerly problematic start states discussed in Exs. 2.2 and 2.3 are both pruned because both can reach $(t1, [2])$ in fewer than 3 context switches: $(t2, [15]) \rightsquigarrow (t1, [13]) \mapsto (t1, [2])$, in the case of Ex. 2.2, and $(t2, [15]) \mapsto (t2, [16]) \rightsquigarrow (t1, [8]) \mapsto (t1, [2])$, in the case of Ex. 2.3.

2. We employ abstract interpretation to remove from consideration some of the states that are not reachable from the program's initial state.

   A proof may fail because it ends up considering states that can never be reached in any execution from the beginning of the program. We postpone a detailed discussion of this phenomenon until §4.2, but note that the issue is another example of Principle 2: we need a method that safely prunes some of the starting states considered during the inductive step. The algorithm in §4.2 addresses this issue by using a simple abstract interpretation that identifies an over-approximation A of the states reachable from the start of the program; all states not in A can be pruned from consideration.

All three techniques—(i) induction over context switches, (ii) pruning à la Claessen's "Improved Induction", and (iii) pruning via abstract interpretation—must be used together to side-step the difficulties encountered in applying K-induction to concurrent programs. We call the combination *K-induction with amplification*.

An important point of comparison between K-induction with amplification and "Improved Induction" relates to guarding against stuttering and, more generally, considering cyclic paths. A cycle automatically defeats an inductive argument because it "eats up" the entire K bound; consequently, others [6, 8, 20] have introduced techniques that, in effect, use *shortest paths* to error to short-circuit longer paths, as well as cyclic paths. In contrast, K-induction with amplification amplifies shortest paths to *shortest execution-context paths*, which are constructed out of unboundedly long execution-context segments.

EXAMPLE 2.4. *Returning to the program from Fig. 1, the inductive case satisfies the conditions of* 3*-induction with amplification:*

> *For each program state, if it is impossible to reach error on any execution with a maximum of 3 context switches, then it is also impossible to reach error from the same starting state in* any *execution that has a maximum of 4 context switches.*

The hypothesis in the inductive case of K-induction with amplification is stronger than that of the inductive case of K-induction because the former asserts that the error state is unreachable for *all* executions sequences that start in a given state and have at most K context switches. In contrast, ordinary K-induction asserts that the error state is unreachable for a *particular* execution sequence σ that has at most K context switches, and it is σ itself that is then *extended* to include one more context switch.

The proof obligation for the inductive case of K-induction with amplification is harder to satisfy than K-induction. To see that, let us rewrite the proof obligation of Exs. 2.1–2.4 in relational notation. (Again, the stacks of the two threads are irrelevant to the point we wish to make, so that portion of the state will be ignored.) The set of possible starting states for the inductive case is

$$\mathbb{S}[pc1, pc2] \quad = \quad \{[2], \ldots, [14]\} \times \{[2], \ldots, [9], [15], [16], [17]\}$$

In the inductive case, we are required to prove

$$\begin{aligned} \forall s \in \mathbb{S}. \quad & \exists p.([2], p) \in R^3(s) \vee (p, [2]) \in R^3(s) \quad \text{[Ind. case (i)]} \\ & \vee \forall p.([2], p) \notin R^4(s) \wedge (p, [2]) \notin R^4(s) \quad \text{[Ind. case (ii)]} \end{aligned}$$

where $R^k(s)$ is the set of states reachable from s within k context switches and $[2]$ denotes *error*. Thus, the formula involves a quantifier alternation.

To finesse this issue, we can restate the proof obligation in terms of (i) backwards reachability from *error*, and (ii) set subtraction

$$B^4(([2], *) \cup (*, [2])) - B^3(([2], *) \cup (*, [2])) = \emptyset, \tag{1}$$

where "*" stands for any program point, and B^k captures backwards reachability in the concurrent program within k context switches. §4.2 describes an approach to checking Eqn. (1) by reachability analysis on a pair of sequential programs obtained from the concurrent program. The first program in the pair simulates a run of the concurrent program for $K + 1$ context switches from an arbitrary state σ; the second program simulates the program for K context switches. For each case, we find the set of all σ that reached error, and see if there are members of the first set $(K + 1)$ that are not in the second (K).

To sum up, the three techniques used in K-induction with amplification have the following benefits:

- Paths through execution contexts are summarized so that arbitrarily long sequences of program actions count as just 1 against the bound of K.
- Instead of considering all start states during the inductive step, states whose shortest execution-context path to error is smaller than the current window are pruned.
- Similarly, some of the states that are unreachable in any execution from the beginning of the program are pruned.

3. K-induction

In this section, we formalize the aspects that were introduced and discussed informally in §2.

3.1 Improved K-induction

We begin with the traditional definition of K-induction for transition systems, and then present a stronger formulation of the principle due to Claessen [6].

DEFINITION 3.1. *A transition system is a 4-tuple $(S, \rightarrow, I, E)$ where S is the set of states, $\rightarrow$ is a binary relation on the set S, and I and E are subsets of S. The elements of set S are called the states, $\rightarrow$ the transition relation, and elements in I and E the initial and error states, respectively.*

Given a transition system $\mathcal{G}$, the problem is either to prove that the states in E are not reachable from the starting states I, or to provide a witness path from some state in I to one in E. The problem is undecidable for infinite-state transition systems. Induction provides a sound but incomplete technique to attempt the proof. We first set up some terminology helpful in presenting the proof techniques. Let $\rightarrow^*$ be the reflexive transitive closure of $\rightarrow$, and $\rightarrow^k_=$ and $\rightarrow^k_\leq$ $(k > 0)$, the relations that capture reachability in exactly k, and $\leq k$ steps, respectively:

$$\begin{aligned} a \rightarrow^k_= b \quad & \overset{def}{:=} \quad \exists a_1, a_2, \ldots a_{k-1} \, : \, (a \rightarrow a_1) \\ & \qquad \wedge (a_1 \rightarrow a_2) \wedge \ldots (a_{k-1} \rightarrow b) \\ a \rightarrow^k_\leq b \quad & \overset{def}{:=} \quad \exists t \, : \, (t <= k) \wedge (a \rightarrow^t_= b) \end{aligned}$$

For a state s, $R(s)$ denotes the set of states reachable from s: $R(s) = \{r \in S | s \rightarrow^* r\}$, and $R(X)$ denotes the set of reachable states from a set X, defined as the union of the sets of states reachable from the elements of X: $R(X) = \bigcup_{s \in X} R(s)$. $R^k_\leq(X)$ and $R^k_=(X)$ are defined similarly.

THEOREM 3.1 (K-induction [6]). *Given a transition system $\mathcal{G} = (S, \rightarrow, I, E)$, E is unreachable from I, i.e., $R(I) \cap E = \emptyset$ if $\exists k \in \mathbb{N}$ such that both of the following hold:*

$$R^k_\leq(I) \cap E = \emptyset \tag{2}$$

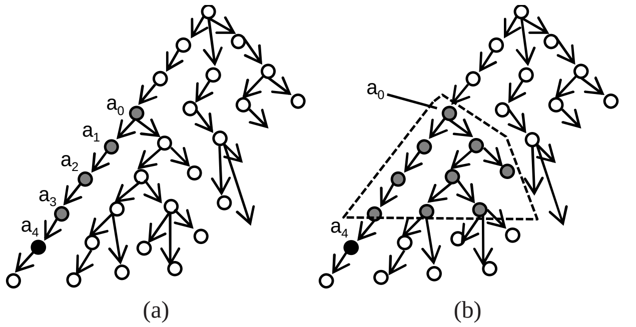

Figure 2. (a) K-induction ($K = 3$): with a_0 as the starting state. The induction step assumes that the hollow nodes on the **path** of length 3 are non-error states and asserts $a_4 \notin E$. (b) Improved K-induction ($K = 3$): with the same starting state, the induction step assumes that the hollow nodes in the **tree** of depth 3 are non-error states (states within the dashed polygon) and asserts $a_4 \notin E$.

$$\forall a_0 a_1 \ldots a_{k+1} \in S : (a_0 \to a_1) \wedge \ldots (a_k \to a_{k+1}) \wedge \quad (\neg a_0 \in E \wedge \ldots \neg a_k \in E) \implies \neg a_{k+1} \in E \tag{3}$$

Obligation (2) states that E is unreachable in K steps from I. Obligation (3) states a property over all paths in the transition system: starting from an arbitrary state, if a path does not reach E in K steps, then it does not reach E at the $K + 1^{st}$ step either.

The modified version of K-induction, called "Improved Induction" in [6], changes the second proof obligation slightly. The hypothesis is strengthened to assume that no path of length up to K from the arbitrarily chosen point (a_0) reaches error. Then, the particular path starting at a_0 (which, by the hypothesis, itself did not reach error in K steps) does not reach E at the $K + 1^{st}$ step.

THEOREM 3.2 (improved K-induction [6]). *Given a transition system $\mathcal{G} = (S, \to, I, E)$, E is unreachable from I, i.e., $R(I) \cap E = \emptyset$ if $\exists k \in \mathbb{N}$ such that both of the following hold:*

$$R^k_{\leq}(I) \cap E = \emptyset \tag{4}$$

$$\forall a_0 \in S \; : \; (R^k_{\leq}(a_0) \cap E \neq \emptyset) \vee (R^{k+1}_{\leq}(a_0) \cap E = \emptyset) \tag{5}$$

Sketch of Proof: We sketch a proof via contradiction. Assume that both obligations (4) and (5) are satisfied, but E is reachable from I. Then there is a path of minimal length from a state $i \in I$ to a state $e \in E$. Let the length of this path be n. Obligation (4) implies $n > K$. The proof proceeds by progressively tiling the steps on the path $i \to^* e$ with trees of depth K disjoint from E and using obligation (5) to infer that the next step of the path extending out from a leaf at depth K of the tree is not in E. Because a path consists of a finite number of transitions, we tile over e eventually, proving $e \notin E$, a contradiction. □

The difference in the second proof obligations of the two versions of induction is brought out in Fig. 2(a) and (b). The figures show a part of the transition graph including an arbitrary node a_0 considered in obligation (3) and obligation (5) for a proof with $K = 3$. In ordinary K-induction (Fig. 2), a path of length 3 from a_0 is assumed to be error free to show that the node a_{k+1} is not an error node. On the other hand, in improved induction (Fig. 2(b)), the tree of depth 3 rooted at a_0 (marked by a dashed polygon) is assumed to be error-free. The stronger induction hypothesis in improved K-induction makes it more expressive. Improved K-induction can often prove a property with a lower bound than ordinary K-induction. Moreover, some properties that are not K-inductive with ordinary K-induction can be proved using improved K-induction.

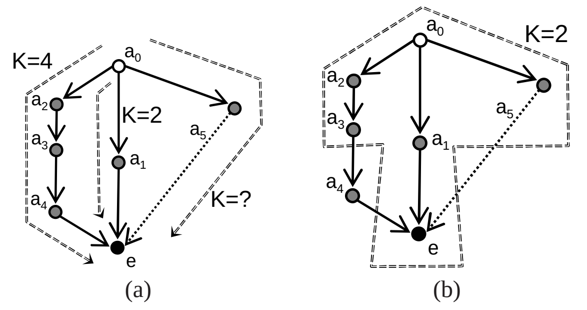

Figure 3. (a) K-induction: e is an error state. The induction obligation vacuously holds along the path $a_0 \to a_1 \to e$ because e is reached within 2 steps from a_0. For the same state a_0, $K \geq 4$ is needed along the path $a_0 \to a_2 \to a_3 \to a_4 \to e$. Along the path $a_0 \to a_5 \to^* e$, no value of K will work if there are paths of unbounded length. The minimum K needed for the obligation to hold for all paths from a_0 is the maximum of the K for all paths. (b) Improved K-induction: In the same graph, the induction obligation holds from a_0 for $K = 2$ because $e \in E$ lies in a tree of depth 2 rooted at a_0. The induction proof works even if there are paths of unbounded length of the form $a_0 \to a_5 \to^* e$.

This difference in strength can be explained in terms of lengths of paths from a state to an error state. Fig. 3(a) and (b) show a subgraph containing a_0 and an error state e. In ordinary K-induction (Fig. 3(a)), proof obligation (3) fails for $K = 1$ on the path $a_0 \rightsquigarrow a_1 \rightsquigarrow e$ because we have $a_0 \to a_1 \wedge a_1 \to e \wedge a_0 \notin E \wedge a_1 \notin E$ but $e \in E$. The minimum value of K for which the proof works is 2, for which the implication in obligation (3) is vacuously satisfied because the antecedent is false. Similarly, obligation (3) holds on the path $a_0 \to a_2 \to a_3 \to a_4 \to e$ for $K \geq 4$ and so forth for different paths from a_0 to e. The minimum value of K needed to prove obligation (3) from a_0 to e is the largest K needed along any path between them. More generally, the minimum value of K needed to prove the second obligation for ordinary K-induction on a transition system is equal to the length of the longest path from a non-error state to an error state.

On the other hand, obligation (5) succeeds from a_0 with $K = 2$. The path $a_0 \to a_1 \to e$ of length 2 falsifies the hypothesis that there is no error state in a tree of depth 2 rooted at a_0, and vacuously proves the obligation (Fig. 3(b)). Longer paths to error from the node do not force K higher because the shortest path to some error state always lies inside a tree of depth larger than the path length. This property of improved induction of short-circuiting longer paths to error by the shortest path is extremely important in the context of concurrent verification. In particular, the path $a_0 \to a_5 \to^* e$ may be of unbounded length, meaning that a counter-example can be found for the second proof obligation of ordinary K-induction for any K. On the other hand, it does not affect the improved K-induction proof and $K = 2$ still suffices to prove obligation (5) from a_0.

LEMMA 3.1 (K-bound). *Let G^E be the subgraph of G backwards reachable from the error set E, and let $I \cap G^E = \emptyset$. Then the property that E is not reachable from I holds in G, and the smallest values of K which a K-induction proof exists, for the two varieties of K-induction, are*

- *Ordinary K-induction: The length of the longest path in G^E.*
- *Improved K-induction:*

$$\max_{a \in G^E \wedge e \in E} \text{length of the shortest path from } a \text{ to } e.$$

3.2 K-induction with amplification

We work with the following notion of a concurrent program: A concurrent program consists of n threads, T_1 through T_n. For each thread, the local state consists of its execution stack—i.e., the values of the local variables in each stack frame—and a shared store S. We assume that the program contains a special error label *err*, and the safety property to be proved is that *err* is unreachable in any execution of the program. A general safety property A can be converted to this form by modifying the program to check A before exit and perform a jump to *err* if the property fails. The semantics of a concurrent program will be modeled formally as a transition system.

We show in this section that K-induction can be modified to include multiple (and possibly an unbounded number of) transitions in each of the K (respectively $K+1$) steps in the proof obligations of the inductive argument. The modification involves a special type of transition system that faithfully models the runtime behavior of the programs.

The idea is formalized in terms of the notion of a *round-robin* schedule. Henceforth, we assume that thread scheduling is round-robin, and call one sweep of the scheduler across all threads an *epoch*. Any execution of the n threads with *arbitrary* interleaving can be modeled by a different execution with round-robin scheduling, provided the threads are allowed to stutter (yield without making progress). Because of the fixed order in which threads are allowed to run during each epoch, it is possible that a larger number of context switches are required to simulate an execution in which there is an arbitrary interleaving of the threads. (In particular, under the round-robin schedule many execution contexts would perform no work before yielding.) However, a round-robin schedule that consists of K epochs is guaranteed to contain all schedules with K or fewer context switches (as well as some schedules with up to nK context switches).

The method of K-induction with amplification developed in the following sections allows us to carry out induction on the number of epochs rather than the number of program steps.

Transition system that corresponds to a concurrent program

For each thread T_j, let $\mathbb{T}^j$ be the local state space and let $\mathbb{S}$ be the set of states of the shared store. Define the graph G^j corresponding to thread j as $G^j = (N_{G^j}, \xrightarrow{G^j})$. Here, $N_{G^j} = \mathbb{T}^j \times \mathbb{S}$ and $\xrightarrow{G^j} \subseteq \mathbb{T}^j \times \mathbb{S} \to \mathbb{T}^j \times \mathbb{S}$ captures the effect of one step of thread j on the state space when run in isolation. To allow the threads to stutter, we also add the transitions $\{(\tau^j, s) \xrightarrow{G^j} (\tau^j, s) \mid \tau \in \mathbb{T}^j, s \in S\}$.

Using the individual transition graphs for the threads $T_1 \ldots T_n$, we define the execution graph for the concurrent program G. The Single Context Graph for thread j, denoted by *SCG*^{j}, captures the semantics of executing thread j without executing a context switch.

DEFINITION 3.2. *SCG*$^j = (N_{SCG^j}, \xrightarrow{SCG^j})$ *with*

$$N_{scg^j} \overset{def}{:=} (\textstyle\prod_i \mathbb{T}^i) \times \mathbb{S} \times \{j\}$$
$$(\tau_1^1, \tau_1^2 \ldots \tau_1^n, s_1, j) \xrightarrow{SCG^j} (\tau_2^1, \tau_2^2 \ldots \tau_2^n, s_2, j) \overset{def}{:=}$$
$$(\forall [i \neq j] \,:\, (\tau_1^i = \tau_2^i)) \wedge ((\tau_1^j, s_1) \xrightarrow{G^j} (\tau_2^j, s_2))$$

The edges of the execution graph can be divided into three types: *SCG* edges ($\xrightarrow{SCG}$) correspond to steps taken by individual threads; Context Switch edges ($\xrightarrow{CS}$) model context switches; and Epoch Switch edges ($\xrightarrow{ES}$) model the step in an epoch in which the last thread in the epoch executes a context switch. We now define the execution graph in terms of these sets of edges.

Using the individual *SCG*s, we define the Single Epoch Graph for the k^{th} epoch, denoted by *SEG*^{k}, as follows; *CS* edges are denoted by (†)

DEFINITION 3.3. *SEG*$^k = (N_{SEG^k}, \xrightarrow{SEG^k})$

$$N_{SEG^k} \overset{def}{:=} (\textstyle\prod_i \mathbb{T}^i) \times \mathbb{S} \times \{1, 2, \ldots, n\} \times \{k\}$$
$$(\tau_1^1, \tau_1^2 \ldots \tau_1^n, s_1, j_1, k) \xrightarrow{SEG^k} (\tau_2^1, \tau_2^2 \ldots, \ldots \tau_2^n, s_2, j_2, k) \overset{def}{:=}$$
$$[((\tau_1^1, \tau_1^2 \ldots \tau_1^n, s_1, j_1) \xrightarrow{SCG^j} (\tau_2^1, \tau_2^2 \ldots \tau_2^n, s_2, j_2)) \wedge (j_1 = j_2)] \quad \vee$$
$$[(\forall i \,:\, (\tau_1^i = \tau_2^i)) \wedge (s_1 = s_2) \wedge (j_1 < n \wedge j_2 = j_1 + 1)] \quad (\dagger)$$

Finally, the execution graph G is obtained by putting together an unbounded number of Round Robin sweeps. In the definition, *ES* edges are denoted by (*).

DEFINITION 3.4. $G = (N, \rightarrow)$

$$N \overset{def}{:=} (\textstyle\prod_i \mathbb{T}^i) \times \mathbb{S} \times \{1, 2 \ldots n\} \times \mathbb{N}$$
$$(\tau_1^1, \tau_1^2 \ldots \tau_1^n, s_1, j_1, k_1) \rightarrow (\tau_2^1, \tau_2^2 \ldots, \ldots \tau_2^n, s_2, j_2, k_2) \overset{def}{:=}$$
$$[((\tau_1^1, \tau_1^2 \ldots \tau_1^n, s_1, j_1, k_1) \xrightarrow{SEG^{k_1}} (\tau_2^1, \tau_2^2 \ldots \tau_2^n, s_2, j_2, k_1)) \wedge (k_1 = k_2)]$$
$$\vee \quad [(\forall i \,:\, (\tau_1^i = \tau_2^i)) \wedge (s_1 = s_2) \wedge ((j_1 = n \wedge j_2 = 1) \wedge (k_2 = k_1 + 1)] \quad (*)$$

This graph, together with $I = \prod_i I^i \times I^S \times \{1\} \times \{1\}$, where I^i and I^S are the possible initial states of the threads and the store, is the transition system that captures all possible executions of the program P. Let us now assume that we also have a set of error states, $E_p \subseteq (\prod_i \mathbb{T}^i) \times \mathbb{S}$. Extend E_p to $E' = E_p \times \{1, 2, \ldots n\} \times \mathbb{N}$. Then, E' is precisely the set of states in G that corresponds to program P reaching an error state in E_p.

We first present a lemma restricting E' to a smaller set E.

LEMMA 3.2. *If a state* $g_1 = (\tau^1, \tau^2, \ldots \tau^n, s, j, k) \in E'$ *is reachable from* I*, then so is* $g_2 = (\tau^1, \tau^2, \ldots \tau^n, s, 1, k+1)$.

As a consequence of Lem. 3.2, we only need to show that the set $E = E_p \times \{1\} \times \mathbb{N}$ is unreachable from I.

DEFINITION 3.5. *Define* $\rightarrow^{\leq k}$ *as an extension of* $\rightarrow$ *in* G *with the property that it contains no more than* k *epoch-switch edges:*

$$\rightarrow^{\leq k} \equiv ((\xrightarrow{SEG})^* \xrightarrow{ES})^{\leq k} (\xrightarrow{SEG})^*$$

With the previous lemma and definition in place, we now state the main theorem.

THEOREM 3.3. *Given an execution graph as defined above,* $\neg(I \rightarrow^* E)$ *holds if both of the following hold:*

$$\neg(I \rightarrow^{\leq k} E) \tag{6}$$

$$\forall s \in \mathbb{T}^1 \times \ldots \mathbb{T}^n \times \mathbb{S} \times \{1\} \times \{1\} \;:\; (s \rightarrow^{\leq k} E) \vee \neg(s \rightarrow^{\leq k+1} E) \tag{7}$$

Proof omitted.

The statement of Thm. 3.3 is similar to Thm. 3.2 from §3.1 in that it breaks down the task of proving that E is unreachable from I into two proof obligations. The proof obligations in Thm. 3.3 differ in that both the base case and the induction case are expressed in terms of an unbounded number of steps (*SEG* edges) in the execution graph. The base case asserts that there is no path containing K or fewer *ES* edges from a state in I to a state in E. This encompasses paths with any number of *SEG* edges between consecutive *ES* edges. In the same spirit, the induction case refers to unboundedly deep trees consisting of paths with respectively K and $K+1$ *ES* edges. The concept of induction over epochs is illustrated in

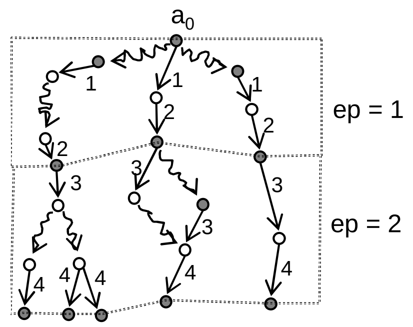

Figure 4. K-induction with amplification: Execution tree for a two-thread program for two epoch steps. Filled circles are states with thread 1 as the active thread and hollow circles are states with thread 2 active. Bold arrows are transitions where a context switch happens (*CS* or *ES* edges). These edges are labeled with the number of context switches since the initial point. Squiggles represent an unbounded number of intra-thread steps (*SEG* edges). One epoch consists of both threads being scheduled once, hence two epoch steps correspond to four context switches. Notice that a $K = 2$ tree in the execution graph is of unbounded depth due to the presence of an unbounded number of *SEG* executed by the active thread in between *CS*/*ES* edges. Each polygon encloses a fragment of the tree covered by one epoch step.

Fig. 4. It shows a part of the execution graph for two threads reachable from a state a_0 within two epoch-steps. Each epoch consists of some number of *SEG* edges, corresponding to program steps executed by the two threads, with an intervening *CS* edge when the first thread yields control to the second thread. The epoch is terminated by an *ES* edge. There is no bound on the number of *SEG* edges in a sequence of *SEG* edges in between two *CS*/*ES* edges. Thus, performing induction with respect to *ES* edges allows us to treat sequences of program steps as a single group that collectively count as just 1 against the bound of K.

Obligation (7) in Thm. 3.3 embodies our response to the two challenges posed in §2.

1. Along with obligation (6), it constitutes a proof that uses paths that consist of many more than K individual program steps by folding longer paths into single epochs.
2. It also effectively guards against reducing to a proof over individual program steps due to the short-circuiting property mentioned in §3.1. An execution from a state s to an error state e with an *ES* edge after every n *SEG* edges can only serve as a counter-example to the proof constructed from K-induction with amplification if there is no other path from s to e with fewer *ES* edges. Hence, an execution trace consisting of $K + 1$ epochs with a context switch after every program point is a counter-example trace only if it is not possible to get from s to e without executing a context switch after every program step. If there were another execution trace from s to e with fewer epoch steps, s would be removed from consideration because the first clause in obligation (7) would be satisfied. Lem. 3.1 guarantees that the minimum K needed for the proof depends on the execution traces that reach an error state with the minimum possible number of *ES* edges; additional traces with more *ES* edges do not push K higher.

4. Proving properties using K-induction with amplification

Thm. 3.3 provides us a technique to prove safety properties for concurrent programs: given a concurrent program with an error label *error*, to prove that *error* is not reachable from the initial state, show that

1. *error* is not reachable from the initial state within K epochs.
2. For every state a_0 of the program, if *error* can not be reached within K epochs then it cannot be reached in $K + 1$ epochs.

Item 1 is the classical problem of Context Bounded Analysis of the given concurrent program with a context bound K. Techniques developed in [4, 13, 17, 18] address this problem. We use a similar sequentialization to reduce item 2 (or equivalently, obligation (7) of Thm. 3.3) to another reachability problem over sequential programs, but it remains a much harder problem because it requires us to establish a property about all paths starting from any arbitrary state in the execution space of the concurrent program.

As noted in §2, we find that an attempt to prove obligation (7) fails if the set from which a_0 is chosen is left unconstrained. Thm. 3.3 already begins by restricting the starting states considered for the obligation to those in the first epoch with the first thread active.

The set of initial states needs to be restricted further to push through a proof of the induction step because interactions between the threads can generate a counter-example to obligation (7).

```
[1] void thread1() {               [5] void thread2() {
[2]     lock();                    [6]     unlock();
[3] }                              [7] }
[4] error: goto error;
```

Figure 5. A program with two threads. Function thread1() is the entry point of the first thread and thread2() of the second thread(). The label *error* is obviously not reachable because there is no jump to *error* from either of the threads.

EXAMPLE 4.1. *Consider the example from Fig. 5. lock() is an atomic function used to acquire a lock. A call to lock() when the lock has already been acquired before it is released using unlock() causes a thread to be blocked. The label error is obviously unreachable because no thread makes a jump to error. However, an attempt to prove obligation (7) with $K = 1$ fails for the program state with the stack configurations (t1, [2][2][4]) (with [4], the error label at the bottom of the stack) and (t2, [6]). The following execution schedule with 2 epochs is a counter-example. (The states are represented by the active thread along with its stack. The third element in each tuple is the current status of the lock: l denotes that the lock has been acquired, while u denotes that it is free.)*

$$(t1, [2][2][4], u) \mapsto (t1, [2][4], l) \rightsquigarrow (t2, [6], l) \mapsto (t2, [\,], u) \rightsquigarrow (t1, [2][4], u) \mapsto (t1, [4], l)$$

Without the context switches, it is impossible to reach error from the state $(t1, [2][2][4], u)$ where $t2$ holds the stack [6]: an epoch is needed to pair up the calls to lock() and unlock() to pop off the top of the call stack for both threads. Indeed, the initial call stacks for the two threads can be extended to arbitrary height to produce counter-examples for any value of K.

We address this problem by pruning out troublesome initial states while proving obligation (7) by only considering states belonging to an over-approximation of the program states that are reachable from the start of the program. To obtain this over-approximation to the reachable state space, we use abstract interpretation to compute the set of all reachable stack configurations for each thread if it were allowed to run in isolation and all branches were explored. The values of local variables in the stack frames and the shared store are left unconstrained.

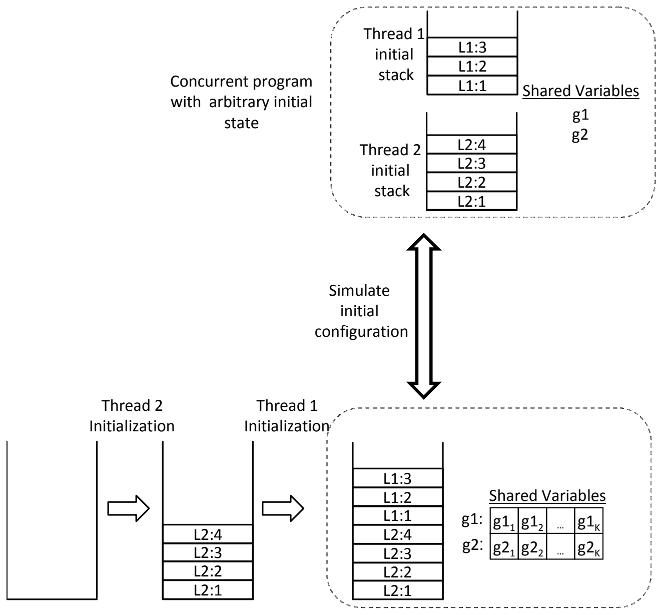

Figure 6. Initialization phase: The first phase synthesizes an arbitrary state of the concurrent program by simulating the execution of each thread running in isolation for some number of steps, and concatenates the stacks obtained at the end of such (incomplete) executions. Shared variables from the concurrent program are represented by K copies in the simulation—one copy for every epoch. Both local variables in the stack and shared variables are assigned random values.

4.1 The general sequentialization technique

This section sketches an approach that uses a sequentialization transformation to convert the original concurrent program into two slightly different sequential versions. We then use reachability queries on the sequential programs to convert obligation (7) into a language-containment query; that is, proving obligation (7) reduces to answering the language-containment query. This general idea yields a source-to-source transformation, described in §4.2, that is applicable to any concurrent program. §4.2 also provides an algorithm to answer the language-containment question for Boolean programs, but leaves open the question for more general programs.

The basic idea is to simulate K-context-bounded executions of the concurrent program starting from an over-approximating envelope of the reachable states, as described above. The simulation involves two phases: an *initialization* phase and a *simulation* phase. First, the *initialization* phase runs code that can nondeterministically generate any one of the states in the envelope of reachable states. The code for the initialization phase simulates each thread in isolation; it assigns random values to all local variables, and ignores all branch conditions. It then concatenates the stacks that were obtained for each thread to obtain a single stack, which serves as the starting stack for the sequential program that runs during the *simulation* phase (Fig. 6).

During the simulation phase, each shared variable from the original program is represented by K copies, one for each epoch. The initial value of the i^{th} copy represents a guess about the value of the variable at the start of the i^{th} epoch. The initial values of the variables are assigned at the end of the initialization phase, where the final step is to assign a random value to all of the copies of the shared variables.

The simulation of K-context-bounded executions of the program involves implementing the possibility of context switches

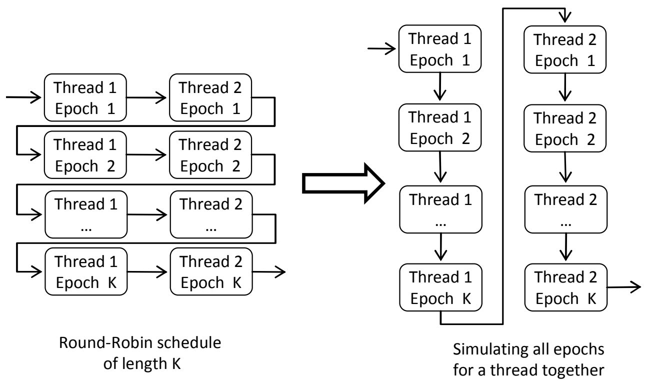

Figure 7. Simulation phase: A round-robin schedule for two threads is simulated by executing all epochs for a thread together [13]. During the i^{th} epoch, the i^{th} copy of the shared variables is accessed.

during the execution of the threads. When a thread yields control, its local state must be remembered while other threads execute, in order for the thread to continue execution from the stored local state. Due to the presence of a stack, there are an unbounded number of local states to remember. This problem is side-stepped by executing all epochs for a thread together (Fig. 7). The thread accesses the i^{th} copy of the shared variables in its i^{th} epoch. When K epochs have been simulated for all threads, the i^{th} copy of a shared variable represents the value of the shared variable at the end of that epoch in the concurrent program. It remains to check that this value matches the guessed initial value for the $i+1^{th}$ epoch.

The simulation phase described above is the same as the sequentialization described in [13] with one minor difference: upon finishing the simulation of all epochs for a thread, we pop off any residual stack for that thread, which uncovers the initial stack (computed during the initialization phase) for the next thread.

Let L^K and L^{K+1} be sequential programs constructed as above to simulate the given concurrent program for K and $K+1$ epochs, respectively. Note that the initialization phases of L^K and L^{K+1} are identical. Let *Envelope* denote the set of states with stack configurations belonging to the restricted set of initial configurations mentioned above.

Let $B^K_{\leq}(\mathit{error})$ be the set of states that are backwards reachable from *error* across K epochs in the concurrent program. Proof obligation (7) can be restated as the following language-containment query:

$$B^{K+1}_{\leq}(\mathit{error}) - B^K_{\leq}(\mathit{error}) = \emptyset. \quad (8)$$

The drawback of using operations that work purely backwards, as in Eqn. (8), is that they can consider states that never arise in any forwards execution. As demonstrated by Ex. 4.1, such states can generate a counter-example to obligation (7). We can do better than Eqn. (8), and in many cases overcome problems like the one discussed in Ex. 4.1, by intersecting each term with *Envelope*

$$(\mathit{Envelope} \cap B^{K+1}_{\leq}(\mathit{error})) - (\mathit{Envelope} \cap B^K_{\leq}(\mathit{error})) = \emptyset. \quad (9)$$

This is where the initialization phase comes to the rescue: all states generated at the end of the initialization phase belong to *Envelope*. Hence, the simulation phase in L^K (L^{K+1}) simulates the concurrent program only from relevant initial states for K ($K+1$) epochs; Eqn. (9) holds iff the set of states at the end of the initialization phase that are backwards-reachable from *error* in L^{K+1} is no greater than the set of states at the same point backwards reachable from *error* in L^K. Thus, obligation (7) can be established by showing that the difference of the set of states at the

```
Simulation functions: func(. . .)                 $\tau_s${stmt}

                                          [1] stmt_tag: $\tau_v${stmt,ep}
func(. . .)             func(. . .)       [2] while(*)
   . . .        ⟹         . . .           [3]     ep = ep + 1;
   stmt                    $\tau_s${stmt}  [4] if(ep > K)
   . . .                   . . .          [5]     if(is_function_main)
                                          [6]       ep = 1;
                                          [7]     return;
```

Figure 8. Transformation of program statements for the simulation phase: All program statements are modified to refer to shared variables for the current epoch (τ_v in the figure); a possible context switch (lines [2]–[3]) and epoch switch (lines [4]–[7]) is implemented immediately thereafter.

```
Main Function : main()                    Base Function : base()

[1] initialize = true                     [7] for(i = 1; i < K; ++i)
[2] last_thread_main_init()               [8]     forall(g ∈ Globals)
[3] for(i = 1; i < K; ++i)                [9]       g[i] = *;
[4]     assume(g_old(i+1) == g(i))        [10]      g_old[i] = g[i];
[5] if(required property fails)           [11] ep = 1;
[6]     goto error;                       BASE:
                                          [12] initialize = false;
```

Figure 9. Additional functions in the transformation: *main* is the entry point of the sequential program. *base* is switching point from the *initialization* to the *simulation* phase.

end of the initialization phase in L^{K+1} and L^K that are backwards reachable from *error* is empty.

4.2 The source-to-source transformation

The simulation described above can be realized in the form of a source-to-source transformation of the concurrent program. The transformed program has two copies of the procedures in the current program: one each for the *initialization* phase and the *simulation* phase. The execution of the transformed program builds up the initial stacks for the threads using the initialization functions. The program then switches to simulation mode to simulate K-context-bounded runs of the concurrent program. The simulation functions are used during the simulating run.

The technique for simulating context-bounded executions is similar to [13]. Fig. 8 shows how the program statements are modified in the functions from the concurrent program to obtain simulation functions. A special variable *ep* stores the current epoch being simulated. All accesses to the shared variables are modified to refer to the copy of the variable for the current epoch. (See the call on operation $\tau_v\{\text{stmt,ep}\}$ in line [1] of Fig. 8.) The program point corresponding to this modified program statement is given a unique label. Variable *ep* is incremented non-deterministically to simulate a context switch (lines [2]–[3]). When *ep* crosses the context-bound, the current thread has been simulated for all epochs, so the function performs an epoch switch by returning immediately (lines [4]–[7]). In fact, when $ep > K$, all the pending calls for this thread also return without performing any additional actions, which uncovers the initial stack of the next thread. For function calls, we assume that returned values are modeled using a special global variable *ret* for each thread. This assumption simplifies the transformation.

Two additional functions in the sequential program tie the initialization and simulation phases together. The functions are shown in Fig. 9. The function *main* is the entry point of the program. It begins the initialization phase by setting a special variable *initialize* to true and calls the initialization copy of the last thread's main function. Control returns to *main* only at the end of the simulation phase. It then performs the check to ensure that the guessed initial values of the shared variables for different epochs were consistent.

```
Initialization functions: func(. . .)                 $\tau_a${stmt}

                                         [1] forall(stmt in statement_list)
func(. . .)             func_init(. . .)    [2]  if(ep > K) return;
   . . .        ⟹        if(initialize)     [3]  if(initialize && *)
   stmt                    forall(l ∈ Locals)  [4]      if(first_thread) base();
   . . .                     l = *;          [5]      else prev_thread_main_init(. . .);
                          . . .              [6]  if(!initialize)
                          $\tau_a${stmt}      [7]      goto stmt_tag;
                          . . .              [8]  $\tau_i${stmt}
```

Figure 10. The initialization functions: Each function from the concurrent program has an initialization copy. These functions build up the stack for the starting state of the simulation. During the simulation phase, control is transferred to the simulation functions. τ_i represents the reinterpreted program statement.

For valid runs, it then checks the property of interest. The function *base* is executed between the two phases. It assigns random values to all copies of the globals and ends the initialization phase by setting *initialize* to false. Function *base* contains the label *BASE*.

The initialization functions mimic an incomplete run of each thread from its entry point, ignoring data, to build up the initial stack for the simulation. The transformation is shown in Fig. 10. All local variables are assigned random values in the header of the function. Data is ignored during initialization by reinterpreting the program statements via the operation $\tau_i\{\text{stmt}\}$ (line [8] in Fig. 10). In $\tau_i\{\text{stmt}\}$, assignment statements are replaced by *skip*; conditionals are replaced by a non-deterministic jump; and function calls are replaced by calls to the initialization copy of the functions. *main* calls the initialization copy of the last thread's main function. The initialization functions for the last thread build up the initial stack for that thread and then call the initialization copy of the previous thread's main and so on (line [5]), eventually calling *base* (line [4]).

The challenge in getting the initialization and simulation phases to work together lies in being able to transfer control to the simulation functions at the right program point during the simulation phase—given that the stacks are initialized with pending return points in the *initialization* code. Line [2] and lines [6]–[7] in Fig. 10 implement the switch from the initialization copy of the function to the simulation copy. Whenever a stack frame from an initialization function is uncovered during the simulation phase, depending on the value of *ep*, the function either returns (line [2]), or jumps to the corresponding program point in the simulation function (lines [6] and [7]). Thus, the initial stack constructed for each thread of the concurrent program in the sequential program consists of activation records for the initialization functions. During the simulation phase, as these stack frames are uncovered, a jump to the corresponding point in the simulation function continues the execution via the simulation functions.

The final transformed problem. We now describe how Eqn. (9) relates to the program transformation presented above. We apply the transformation twice to P to obtain the sequential programs L^K and L^{K+1}, which simulate P for K and $K+1$ epochs, respectively. $(\textit{Envelope} \cap B^K_{\leq}(\textit{error}))$ represents the set of states at the label BASE (line [12] in Fig. 9) in L^K that are forwards reachable from the program entry and backwards reachable from *error*. Similarly, $(\textit{Envelope} \cap B^{K+1}_{\leq}(\textit{error}))$ represents the set of states at BASE in L^{K+1} that are forwards reachable from the program entry and backwards reachable from *error*. It remains to compute these sets and take their difference.[2] Thus, the proof obligation of the inductive case leaves us with the following alternative problem:

[2] Technically, L^K and L^{K+1} must have identical state spaces. This issue is handled by declaring an extra unused copy of the globals in L^K.

- Given the two sequential programs L^K and L^{K+1} with identical state spaces, and special points *error* and BASE in both programs, check whether there is a program state in the forwards-reachable set at BASE that is backwards-reachable from *error* in L^{K+1}, but not backwards-reachable from *error* in L^K.

The program transformation defined above reduces the proof obligation of the inductive case from a reachability problem on a concurrent program to this reachability problem on a pair of sequential programs. For general programs, we do not know of a solution to the reformulated problem. For the case of recursive Boolean programs, the question can be addressed via reachability queries on pushdown systems.

DEFINITION 4.1. *A pushdown system (PDS) is a triple* $\mathcal{P} = (P, \Gamma, \Delta)$ *where* P *is a set of states,* Γ *is a set of stack symbols, and* $\Delta \subseteq \mathcal{P} \times \Gamma \times \mathcal{P} \times \Gamma^*$ *is a finite set of rules. Elements from the set* $P \times \Gamma^*$ *are called configurations of the PDS. A PDS rule is written as* $(p, \gamma) \hookrightarrow (q, u)$ *where* $p, q \in \mathcal{P}$, $\gamma \in \Gamma$ *and* $u \in \Gamma^*$. *The rules define a transition relation* $\Rightarrow$ *on configurations of* $\mathcal{P}$*: If* $(p, \gamma) \hookrightarrow (q, u)$ *then* $\forall u' \in \Gamma^* (p, \gamma u') \Rightarrow (p, uu')$. *The reflexive transitive closure of* $\Rightarrow$*, denoted by* $\Rightarrow^*$*, is the reachability relation defined by the runs of the PDS over the configuration space.*

A PDS can be used to model the control flow of a program. The PDS has a single state p and stack symbols corresponding to the program points. An intraprocedural edge (u, v) is modeled by the PDS rule $(p, u) \hookrightarrow (p, v)$; a call to a function with entry e, from the call-site c, and with a return to r, is modeled by the rule $(p, c) \hookrightarrow (p, er)$; and a return statement from program point r is modeled by the rule $(p, r) \hookrightarrow (p, \epsilon)$. Data in the Boolean program is encoded by expanding P to be the set of global states and expanding the stack symbols to store the values of the local variables. The PDS rules are also updated to reflect the effect of the program steps on the values of the global and local variables.

Given a Boolean program to verify, applying the source-to-source transformation explained above yields the two program simulations for K and $K+1$ epochs. Abusing notation, we use L^K and L^{K+1} to refer to the PDS models of the two sequential programs obtained. Because the procedure *base()* had no local variables in the source-to-source transformation, the stack symbols for the procedure correspond one-to-one with the program points. We use BASE to denote the stack symbol corresponding to the program label with the same name. Similarly, let ERR be a special stack symbol corresponding to the error point in *main()* (line [6] of Fig. 9).

There are two main operations needed to verify the induction step using L^K and L^{K+1}. Given the two PDS models $L^K = (P^K, \Gamma^K, \Delta^K)$ and $L^{K+1} = (P^{K+1}, \Gamma^{K+1}, \Delta^{K+1})$,

1. Calculate M^K: the set of configurations that arise in the runs of the PDS L^K with BASE at the top of the stack, that are backwards reachable from the configurations with ERR on top of the stack. Similarly, calculate M^{K+1}.
2. Check language containment: $L(M^{K+1}) \subseteq^? L(M^K)$

K-induction with amplification for a given K is tractable for Boolean programs because both items 1 and 2 are computable. For a regular set of configurations Q in the configuration space of a PDS, the set of configurations reachable in the runs of the PDS starting from configurations in Q is also a regular set. There are algorithms $post^*$ and pre^* [3] that compute the forwards (respectively, backwards) reachable set of configurations from a regular set of configurations.

$$post^*[\mathcal{P}](Q) = \{v \in \Gamma^* \mid \exists u \in Q \quad u \Rightarrow^*_{\mathcal{P}} v\}$$

$$pre^*[\mathcal{P}](Q) = \{v \in \Gamma^* \mid \exists u \in Q \quad v \Rightarrow^*_{\mathcal{P}} u\}$$

Both $post^*$ and pre^* answer queries about an input set given in the form of a finite state automaton (FSA). A configuration (p, u) is in the set represented by automaton Q iff Q accepts u starting from the start state p. The answer is returned in the form of a similar automaton.

Item 1 above can be calculated as

$$M^K = post[L^K]^*(I) \cap \mathcal{A}^{BASE} \cap pre[L^K]^*(\mathcal{A}^{ERR}).$$

Here $\mathcal{A}^{BASE}$ is an FSA that accepts configurations $\{(p, BASE\ u) \mid p \in P^K \wedge u \in (\Gamma^K)^*\}$; $\mathcal{A}^{ERR}$ is an FSA that accepts configurations $\{(p, ERR\ u) \mid p \in P^K \wedge u \in (\Gamma^K)^*\}$; and I is the set of initial configurations of the PDS $\mathcal{P}$. Note that the subterm $post[L^K]^*(I) \cap \mathcal{A}^{BASE}$ intersects an automaton that accepts all configurations forwards reachable from the initial point ($post[L^K]^*(I)$) with an automaton that accepts configurations with BASE at the top of the stack ($\mathcal{A}^{BASE}$), and hence corresponds to *Envelope* in Eqn. (9).

The check performed in item 2 is standard regular-language containment.

Algorithm 1 Semi-decision procedure for Boolean programs.

```
1:  // $\mathcal{A}^{BASE}$ is an automaton that accepts $BASE\ \Gamma^*$
2:  // $\mathcal{A}^{ERR}$ is an automaton that accepts $ERR\ \Gamma^*$
3:  for $K = 1$; true; $K = K + 1$ do
4:      if CBA$^K$(P) returns reachable then
5:          // Error is reachable: bug found
6:          EXIT
7:      // Compute the set of states that reach error in $K$ epochs, $M^K$
8:      synthesize $L^K$ from P
9:      $M^K$ = post[$L^K$]*(I) ∩ $\mathcal{A}^{BASE}$ ∩ pre[$L^K$]*($\mathcal{A}^{ERR}$)
10:     // Compute the set of states that reach error in $K + 1$ epochs, M$^K + 1$
11:     synthesize $L^{K+1}$ from P
12:     $M^{K+1}$ = post[$L^{K+1}$]*(I) ∩ $\mathcal{A}^{BASE}$ ∩ pre[$L^{K+1}$]*($\mathcal{A}^{ERR}$)
13:     if $M^{K+1} - M^K = \emptyset$ then
14:         // Both obligations passed: program proved correct
15:         EXIT
```

The semi-decision procedure. Alg. 1 states a semi-decision procedure for recursive concurrent Boolean programs based on the ideas discussed above. The algorithm starts with $K = 1$ and tries to push through a proof using K-induction with amplification. The base case is checked by applying CBA [13] with the context bound K. The inductive step is checked via sequentialization and the language-containment operation described above. If ERR can be reached in a run of the concurrent program within K epoch switches, CBA returns a trace for the run. The algorithm exits with the error trace: a bug has been found.[3] If the program passes both CBA and the containment check, ERR is unreachable and a K-induction with amplification proof has been found for the bound K. If the program passes CBA, but the containment check fails, K is incremented and the loop repeats.

For the class of non-recursive, concurrent Boolean programs, Alg. 1 is a decision procedure. If the program has a bug, there exists an error trace within a finite number of epochs K, and CBA fails for that context bound. If the program has no bug, there is some K for which both checks in Alg. 1 will succeed: non-recursive, concurrent Boolean programs have a finite number of states, and Alg. 1 does not explore cyclic paths (paths with repeated states), which implies that it will eventually terminate.

In addition to non-recursive, concurrent Boolean programs, we identified a few classes of recursive, concurrent Boolean programs for which Alg. 1 is a decision procedure. One class consists of recursive Boolean programs in which all reads of the shared variables occur only when the stack is no deeper than some fixed bound.

[3] If the Boolean program is an abstraction of some concrete program, and we wish to use Alg. 1 inside an abstraction-refinement loop, line 5 is where the counter-example would be checked to see whether it is spurious, and if so, a refined Boolean program would be generated.

```
[1] bool x;
[2] bool y;
[3]error: goto error;

[4] void thread1(){
[5]     if(*)
[6]       thread1();
[7]     while(x == true){}
[8]     x = true;
[9]}

[10] void thread2(){
[11]     if(*)
[12]       thread2();
[13]     while(x == false){}
[14]     x = false;

[15] void main2(){
[16]     y = false;
[17]     thread2();
[18]     if(y == true) goto error;
[19]}
```

Figure 11. Limitation of K-induction with amplification: A program that is not K-inductive for K-induction with amplification.

In such programs, functions that read global variables are non-recursive and cannot be called (even transitively) by recursive procedures. Recursive procedures are free to write to global variables. We omit a proof of this result for lack of space.

5. Limitations of K-induction with amplification

Any method proposed as a verification technique for concurrent programs is inherently incomplete because the problem of proving safety properties is undecidable, even for recursive concurrent Boolean programs. In the case of K-induction with amplification, the minimum value of K needed for obligation (7) in the K-induction proof to work is bounded from below by the length of the shortest execution-context path from a program state to error. While we restrict the set of states considered as the initial states using abstract interpretation techniques, this pruning of the states is not enough to guarantee that a proof can be found for any K. Indeed, due to the inherent undecidability of the problem, there is *no* way to guarantee that such a K can be found.

The program shown in Fig. 11 eludes proof by K-induction with amplification. The program has two threads with entry functions *thread1()* and *main2()*. The execution of either thread consists of an arbitrary number of recursive calls (lines [5]–[6] and lines [11]–[12]), followed by returns down to the first caller. The *while* loop in thread 1 blocks the thread until thread 2 sets x to *false* and the *while* loop in thread 2 blocks until thread 1 sets it back to *true*. Consequently, the two threads must return alternately. *error* is not reachable because y is set to false before the call to *thread2()* and remains false at line [2].

Unfortunately, it is not possible to prove that *error* is unreachable using K-induction with amplification. The difficulty lies in the forced synchronization before every return in the program. From some arbitrary starting stack configurations for the two threads, our method must expend one epoch to reduce the stack size by 1. Because the initial stacks can be of any depth, obligation (7) fails for any value of K. (We encountered this problem earlier in the example in §4 (Fig. 5), although the problematic stack configurations there were spurious.) In particular, the following is a scheme for generating initial states for which the second obligation fails for a general K: (x = *true*, y = *true*, ([7]$^{K+1}$, [13]$^{K+1}$[18])). It is not possible for the second thread to finish all $K+1$ calls to *thread2()* within K epochs. Using $K+1$ epochs, there is a schedule for which thread 2 reaches the if condition in line [18], and because y is assumed to be *true*, thread 2 reaches *error*. The stack configurations in these initial states are indeed reachable in the program—albeit with different data states—and thus they are not pruned by the abstract-interpretation based method for removing spurious starting states.

6. Experiments

To test the capabilities of K-induction with amplification, we ran it on a small corpus of examples (whose characteristics are listed in columns 2–7 of Fig. 12). The experiments were performed on a system with dual quad-core, 2.82 GHz Intel processors; however, the analyzer's implementation is single-threaded. The system has 6 GB of memory and runs 64-bit Linux (Red Hat 2.6.18).

Program (Set)	#threads	LOC	#globals	#locals	K	Time(s)
blue2 (Blue)	2		5	6	3	32.44
dekkers (Sync)	2		4	1	3	3.74
lock_unlock_K (Sync)	2		2	1	4	0.6
lock_unlock_3 (Sync)	3		4	3	3	44.02
petersons (Sync)	2		4	1	3	0.3
petersons_loop (Sync)	2		4	1	3	6.05

Figure 12. Summary of the experiments performed using K-induction with amplification for non-recursive Boolean programs. Column 1 reports the name, along with the name the program suite: Sync is a set of simple mutual-exclusion examples we used for benchmarking; Blue refers to a set of Bluetooth-driver examples [11]. Columns 2–7 report the number of threads, total lines of code, globals, locals, the minimum bound K needed for the proof to work, and the time needed for the proof with bound K.

(a)

K	4	5	6	7	8
Time(s)	0.64	1.90	5.69	19.57	108.71

(b)

#Threads	2	3	4	5	6	7
Time(s)	0.03	0.27	2.17	17.09	132.24	1011

Figure 13. (a) K vs. time: The effect of increasing the K needed for a proof on the time taken. Programs belong to the *PumpK* set. (b) Number of threads vs. time: The effect of varying the number of threads on time taken for a proof with $K = 3$ for a simple mutual-exclusion program.

The experiments were designed to test the following questions:

- What are the typical values of K needed for a proof?
- How does the method behave as K is increased (for a program of essentially fixed size)?
- How does the method scale as the number of threads is increased?

Each example consisted of two or more threads with a distinguished point that the algorithm attempts to establish is unreachable.

The experiments involved three sets of programs.

blue2: A model of a Bluetooth driver [11] that has been used in several earlier studies [5, 18].

Sync: Implementations of several mutual-exclusion algorithms, each involving two or three threads. One of the examples from this set was used to study the effect of varying the number of threads competing for the critical section.

PumpK: These programs push the minimum K up by introducing many synchronization points between the threads. We used the programs to study the effect of increasing K on the analyzer.

We implemented a variant of Alg. 1 by extending a previous implementation of CBA [13], which was built on top of the Moped [19] model checker for pushdown systems (PDSs). Moped provides a modeling language, called Remopla, which it compiles to Boolean programs. Instead of performing a source-to-source transformation, as described in §4.2, our implementation uses the PDS rules that Moped generates from a Rempola model as an intermediate representation (IR), and performs an IR-to-IR transformation. We decided to use Moped because the input language simplifies specifications of our target programs, and because Moped supports backwards reachability queries on the program model generated; The ability to answer backwards reachability queries is crucial for the semi-algorithm developed in §4. [4]

[4] The Remopla models used in the experiments are available for download at http://pages.wisc.edu/∼pprabhu/kindamp_exp/

The current implementation falls short of demonstrating the full capabilities of Alg. 1. Although Alg. 1 is applicable to a wider class of programs, the implementation handles only non-recursive Boolean programs. In the absence of recursion, the set of stack configurations reachable from the start state is finite, which means that for Boolean programs, there are only finitely many different program states that can arise in any execution of the programs. Thus, the problem of proving that a set of states is not reachable from the initial state is decidable; a brute-force computation of the reachable state space always terminates. On the other hand, focusing on non-recursive programs lets us validate the applicability of the induction principle easily, albeit on a small scale.

With respect to the questions posed at the beginning of this section, we made the following observations:

Typical values of K. With the exception of the programs reported in Fig. 13(a), Alg. 1 found a proof for all programs with K = 3 or 4. The programs used in Fig. 13(a) were specifically constructed to pump K up. This emperical measurment shows that K-induction with amplification greatly reduces the bound K needed for a proof. The minimum K for which a simple induction proof can be found is roughly proportional to the length of the program text.

Behavior as K increases. We found that the time taken for the induction step of the proof increases rapidly with increasing value of K. The number of basic PDS operations performed as K increases (not reported) remains roughly the same. The slowdown is a result of the increase in the size of the data structures used, which makes the PDS operations used in the implementation expensive.

Behavior as the number of threads increases. The final subtraction operation in Alg. 1 is implemented in a naive fashion, which leads to an exponential increase in time with the number of threads.

7. Related Work

The goal of verifying properties of concurrent software via model-checking techniques has a long history, going back to the origin of the field [7].

K-induction [1, 6, 20] has been studied in the hardware model-checking community for analysis of circuits, as well as in the software-verification community for analysis of sequential programs [9]. As discussed in §2 and §3, we adopt Claessen's method of "Improved Induction" except that, following Principle 1 of §2, induction is performed over epochs rather than program steps.

The first work to use K-induction to analyze concurrent software was by de Moura et al. [8]. Both our work and that of de Moura et al. share the goal of using K-induction to augment a core under-approximating method—which may fail to explore some behaviors of a program—to make it possible to verify properties. In their case, they combined K-induction with bounded model checking (BMC) [2]; in our work, we combine K-induction with CBA [13, 17]. Applying K-induction in conjunction with CBA comes with its own set of challenges. The techniques that we used to address these challenges have been described in §2–§4.

De Moura et al. make use of a technique that appears to be an independent rediscovery of Claessen's "Improved Induction":

> . . . whenever the induction step . . . fails . . . we define the predicate $U(s)$ for representing the set of . . . states [that] may reach the bad state in k steps . . . Now φ is strengthened as $\varphi \wedge \neg U(s)$, and quantifier elimination is used for transforming this strengthened formula into an equivalent Boolean constraint formula [8, §5].

Thus, de Moura et al. incorporate "Improved Induction" using a blocking conjunct, whereas we incorporate it via the set-subtraction operation of Eqn. (9).

One major difference between the two techniques is that our approach performs K-induction over epochs. Although de Moura et al. use two kinds of simulation relations to reduce the size of the state space they work with, in essence they still perform induction over program steps. Their path-compression techniques ("direct simulation" and "reverse simulation") are fairly weak—basically variants on "paths of interest contain no repeated states", such as "paths of interest contain no repeated states, modulo differences in the values of the (write-once) input variables". In contrast, the technique of K-induction over epoch steps permits some arbitrarily long sequences of program actions to be condensed so that they count as just 1 against the bound of K.

The idea of creating analyzers that place a bound on the number of context switches that a multi-threaded program is allowed to perform (i.e., CBA) originates in the work of Qadeer and Wu [18]. Qadeer and Rehof [17] showed that CBA is decidable for Boolean programs. Bouajjani et al. [4] extended the decidability result to Boolean programs with bounded heaps, and Lal et al. [14] extended the result to a class of infinite-state program abstractions. All of these techniques bound the number of context switches that are explored (while letting processes perform an arbitrary number of computation steps in between context switches). Although CBA under-approximates the program's semantics (i.e., it may fail to explore some behaviors of the program) it encompasses a large, and in general unbounded, subset of the program's behaviors. In particular, CBA does not impose any bound on the execution length between context switches.

The use of a sequentialization transformation to reduce a CBA problem to a (larger) sequential-analysis problem in which portions of the state are replicated was pioneered by Qadeer and Wu [18], although the Qadeer-Wu transformation is limited to a fixed context bound of 2. The first sequentialization transformation that allowed for an arbitrary context bound K was given by Lal and Reps [13]. A different sequentialization transformation for CBA-K was given by La Torre et al. [12]. Although the transformed program increases in size with the sequentialization approach, the significance of these methods is that they allows any sequential-analysis technique to be applied after the transformation. Moreover, they have led to order-of-magnitude speed-ups over some other competing approaches for analyzing concurrent software [13].

Another recent approach to verifying concurrent programs has been described by Garg and Madhusudan [10]. Their technique applies to programs for which a *rely-guarantee* proof exists for the property of interest. The class of programs addressed by our technique is incomparable to the class addressed by Garg and Madhusudan, and the two methods are based on quite different approaches. Both techniques are of interest for expanding the tools available for addressing the difficult and important problem of verifying concurrent software.

References

[1] R. Armoni, L. Fix, R. Fraer, S. Huddleston, N. Piterman, and M. Vardi. SAT-based induction for temporal safety properties. *ENTCS*, 119(2), 2005.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, 1999.

[3] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*. 1997.

[4] A. Bouajjani, S. Fratani, and S. Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In *CAV*. 2007.

[5] S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *TACAS*, 2006.

[6] K. Claessen. Induction and state machines. In *Winter Meeting of the Computing Science Dept.* Chalmers Univ. of Tech., Sweden, 1999.

[7] E. Clarke, Jr. and E. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Workshop on Logic of Programs*, 1981.

[8] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *CAV*. 2003.

[9] A. Donaldson, L. Haller, D. Kroening, and P. Ruemmer. Software verification using k-induction. In *SAS*, 2011.

[10] P. Garg and P. Madhusudan. Compositionality entails sequentializability. In *TACAS*. 2011.

[11] N. Kidd. The Bluetooth driver models, 2009. http://pages.cs.wisc.edu/∼kidd/bluetooth.

[12] S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV*, 2009.

[13] A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *FMSD*, 35(1), 2009.

[14] A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS*, 2008.

[15] L. Lamport. What good is temporal logic? In *IFIP World Congress*, 1983.

[16] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[17] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.

[18] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *PLDI*, 2004.

[19] S. Schwoon. Moped system. http://www.fmi.uni-stuttgart.de/szs/tools/moped/.

[20] M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, 2000.