Reducing Context-Bounded Concurrent Reachability to Sequential Reachability*

Salvatore La Torre[1], P. Madhusudan[2], and Gennaro Parlato[1,2]

[1] Università degli Studi di Salerno, Italy
[2] University of Illinois at Urbana-Champaign, USA

Abstract. We give a translation from concurrent programs to sequential programs that reduces the context-bounded reachability problem in the concurrent program to a reachability problem in the sequential one. The translation has two salient features: (a) the sequential program tracks, at any time, the local state of only one thread (though it does track multiple copies of shared variables), and (b) all reachable states of the sequential program correspond to reachable states of the concurrent program.

We also implement our transformation in the setting of concurrent recursive programs with finite data domains, and show that the resulting sequential program can be model-checked efficiently using existing recursive sequential program reachability tools.

1 Introduction

Analysis of concurrent programs is an important problem that is difficult for a variety of reasons. The explosion in the number of interleavings between threads is one problem, and the explosion in the state-space needed to keep track of the combined states of each thread is another. For instance, even checking reachability of n parallel systems, each modeled as a *finite-state* transition system (with some form of communication) is solvable only in time exponential in n, as there are exponentially many global configurations which are feasible.

In this paper, we consider the problem of translating concurrent programs to sequential programs, which reduces the reachability problem for the former to that for the latter. The motivation behind such a translation is to use the fairly sophisticated sequential analysis tools to analyze concurrent programs. For instance, this has been proposed in this same volume by Lahiri, Qadeer, and Rakamarić [5], in order to apply deductive verification tools based on SMT solvers to verify concurrent C programs.

A translation of the above kind is of course always possible— given a concurrent program, we can build a sequential program that simply simulates its (global) behavior. However, such a sequential interpreter would track the entire global state of the concurrent program, which involves keeping the local state of each thread. Our aim is to provide a translation that avoids this extreme blow-up

* This work was partially funded by NSF CAREER Award CCF 0747041, by the MIUR grants ex-60% 2007-2008, and FARB 2009 Università degli Studi di Salerno (Italy).

A. Bouajjani and O. Maler (Eds.): CAV 2009, LNCS 5643, pp. 477–492, 2009.

in state-space, and build a sequential program that tracks, at any point, the local state of only one thread of the concurrent program. However, such a translation is not always feasible.

A restricted reachability for concurrent programs has emerged in the last few years, mainly motivated from testing concurrent programs and model-checking finite-state models of them, called *context-bounded reachability*, wherein we ask whether an error state is reachable within k context-switches, for a fixed k. This was first suggested by Qadeer and Wu [11], and has several appealing features.

First, it has been argued that bounded context-switching is a natural restriction as most concurrency related errors manifest themselves within a few context-switches. Musuvathi and Qadeer have, for example, experimentally shown that a few context-switches explores a vast space of reachable configurations [9].

Second, when we deal with concurrent programs where variables range over finite domains (obtained either by restricting the domain or using some form of abstraction such as predicate abstraction), we obtain essentially concurrent pushdown systems, which have an undecidable reachability problem. However, it turns out, mainly due to results of [10], that the bounded context-switching reachability problem is decidable.

Finally, and perhaps most importantly, when one examines why context-bounded reachability is decidable for concurrent programs over finite data domains, the primary reason is that we can *compositionally* analyze the program, examining each thread *separately* and combining the results. This compositional reasoning hence involves searching a state-space where at any point only the local state of a single thread is tracked.

In [6], Lal and Reps propose a transformation of concurrent programs to sequential programs that reduces context-bounded reachability in the former to reachability in the latter. This translation exploits compositional arguments underlying the decidability proofs to construct a sequential program that tracks, at any point, the local state of one thread, the shared state, as well as k copies of the shared state (corresponding to the shared state at the k context-switches). This translation is appealing when the local state is complex (in particular, when the local state has a stack to model recursive control), and the shared state is comparably less complex.

The translation proposed by Lal and Reps, however, does not permit a *lazy* analysis: the sequential program *guesses* in advance the valuations of shared variables $g_1, \ldots, g_k$ at the context-switches, and verifies each thread locally against this guess. Hence, the sequential program unnecessarily explores unreachable states of the concurrent program (as the guessed g_i's may not be reachable).

In our opinion, a transformation that results in a lazy analysis (one which explores only reachable states) is highly desirable; for example, in model-checking, it can drastically reduce the size of the state-space that needs to be explored. In fact, Lal and Reps [6] do give direct lazy analysis algorithms for finite-state programs, and our recent work in [4] also provides a direct fixed-point algorithm for lazy analysis. However, a transformation from concurrent programs to sequential programs that preserves laziness was not known.

Contributions. In this paper, we show a lazy translation: given a concurrent program and a bound k, we show how it can be transformed to a sequential program, such that reachability within k context-switches in the concurrent program reduces to reachability in the sequential one. Moreover, the salient feature of the translation is that the reachable states of the sequential program correspond to reachable states of the concurrent program, and hence is, in our opinion, a more faithful representation of the sequential program. The main idea behind our reduction is to have the sequential program *calling individual threads multiple times from scratch* in order to recompute the local states at context-switches.

We also implement an eager translation and our lazy translation for concurrent programs over finite data-domains (a.k.a. concurrent Boolean programs). This results in sequential programs over a finite data-domain, which can be model-checked using existing tools. Our implementations of the translations are available online[1]. We show that our laziness-preserving transformation outperforms eager transformations on a class of multithreaded Bluetooth driver examples (the original programs and the transformed ones are also available online).

The paper is organized as follows: Section 2 gives a high-level and intuitive description of our eager and lazy translations; Section 3 formally defines the class of sequential and concurrent programs; Section 4 describes the eager translation (which is mainly adapted from ideas in [6]); the laziness-preserving translation is presented in Section 5; Section 6 reports on our implementation and experiments for Boolean programs; and Section 7 concludes with some future directions.

Related Work. Bounded context-switching reachability was introduced in [11]: the KISS project implemented reachability within two context-switches and found data-race errors in device drivers; interestingly, it also reduced the problem to sequential reachability (the reduction is simple for two switches). Decidability of context-bounded analysis for concurrent recursive Boolean programs was established in [10] using automata theoretic methods. There have been a number of analysis algorithms and implementations of context-bounded reachability problems: [13] implement the automata-theoretic solution symbolically, [6] propose an algorithm to compute the reachable states lazily, and the work in [4] implements symbolic fixed-point based solutions for lazy reachability. Bounded context-switching has also been exploited in other contexts: the tool CHESS [8] explores bounded context-switching interleavings to test concurrent programs, and bounded context-switching for systems with heaps [1], systems communicating using queues [3], and weighted pushdown systems [7] have been proposed.

Bounded context-switches vs bounded rounds. Lal and Reps [6], apart from giving a transformation for eager analysis, also make a technical improvement: by using only k extra sets of shared variables, we can explore all the state space reachable in k round-robin rounds (which is larger than that reached in k context-switches).

In both our transformations, we have considered bounded context-switching reachability, and not bounded round-robin reachability (this is why our eager transformation is slightly different from that of [6]). The reason is that we see no

[1] At `http://www.cs.uiuc.edu/~madhu/getafix/cbp2bp`

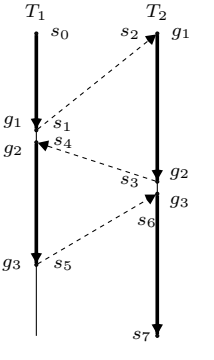
efficient way of transforming programs using only $O(k)$ sets of global variables that effects a *lazy-transformation*. While we do know of such a translation, this requires calling each thread too many times (exponentially) many times, which we believe will not work well in practice. This translation is out of the scope of this paper, and the problem of coming up with a more efficient translation for bounded rounds reachability is an open problem.

2 From Concurrent to Sequential Programs

In this section, we briefly sketch two translation schemes from concurrent to sequential programs which preserve reachability up to a fixed number of context-switches. The transformations work for general programs (even when the domain of variables is not bounded), except that they assume that the state of shared variables is known, and can be replicated and compared against each other.

Of course, there is a simple translation of concurrent programs to sequential programs that keeps the entire global configuration of the program. However, our aim here is to build a sequential program that keeps only the local state of a single thread at any time (though we allow some copies of shared states). In particular, when applied to concurrent recursive programs over *finite data-domains*, the translation will should yield a sequential recursive program over a finite data-domain; this would reduce concurrent context-bounded reachability on finite domains to reachability of sequential programs over finite domains, which is a decidable problem.

We consider concurrent programs with a *fixed* number of threads that communicate with each other using shared variables. Each thread is a sequential program (with possibly recursive procedures). A state is thus given by the values of the shared variables and the local state of each thread. A computation of a concurrent program is a sequence of contexts, where in each context a single thread has control.



For ease of presentation let us first consider the case of a concurrent program with only two threads T_1 and T_2. Also, instead of describing the transformation, we take one particular path in the concurrent program, and describe how the corresponding sequential program will discover it.

Consider a sample computation whose control flow at the thread level is shown on the right. It starts in a state s_0 with the control in T_1, then at a state s_1 switches the control to T_2 at state s_2, then locally executes T_2 till it reaches the state s_3, and then switches to T_1 again at state s_4, and so on. In a simple sequential simulation of this computation, while we execute instructions of T_2, say from s_2, we need to remember the local component of thread T_1 in state s_1, in order to compute the switch from s_3 to s_4. This can be very expensive. In the case of finite data domains, where threads have recursive procedures and we would like to build a sequential program also over a finite data domain, this is in fact *impossible*, as it

involves keeping the unbounded call-stack of T_1 at s_1. Schemes for translation to sequential programs hence need to explore this run in a different manner, where only the local state of one process is kept at any given point.

The two translation schemes we present in the rest of this section do not store the local states of more than one thread, but will store k valuations of shared variables, which intuitively correspond to the value of shared variables at context-switches. In the computation sketched above, if g is the set of shared variables, we denote by g_1 the value of the shared variables at the first context-switch from T_1 to T_2, i.e., the value of shared variables at states s_1 and s_2. Similarly, let g_2 be the value at the next context-switch back to T_1 (i.e., at states s_3 and s_4), and g_3 the value at the last context-switch back to T_2 (i.e., at states s_5 and s_6).

The eager and lazy translations differ in the way they compute the values of these shared variables at the context-switches: the eager approach *nondeterministically* guesses them right at the beginning, while the lazy approach computes them dynamically.

The eager approach. In the eager approach, we guess the values g_1, g_2, g_3 at the beginning (non-deterministically). Then, we process thread T_1 completely, handling both segments of its run (s_0 to s_1 and s_4 to s_5), and then erase the local state and proceed to process the two segments of thread T_2 (s_2 to s_3 and s_6 to s_7). Note that each thread is processed only once.

More precisely, after guessing g_1, g_2 and g_3, we start to compute in T_1 the first context. At any point where the shared variables match the guessed value g_1, we allow a "jump" where the global variables are rewritten to g_2, and we proceed in T_1 computing the *third* context, till we reach g_3. Note that across the "jump" above (from s_1 to s_4), the local state of thread T_1 is preserved. Hence the call to thread T_1 verifies that there is a run which can reach shared-variable-state g_1, jump to g_2, and proceed to reach g_3.

Next, we erase the local state of T_1 and proceed to process thread T_2 from its initial local states with the shared variables assigned with g_1. We then continue till we reach a state with shared-variable valuation g_2, jump to g_3 preserving local state, and continue the computation of the fourth context in T_2.

Intuitively, the guessed values g_1, g_2, g_3 allow the stitching of the two execution segments that have been executed in the two threads. However, since the tuple of values of the shared variables is guessed, they may not correspond to reachable states of the concurrent program, and hence may lead to exploring the individual threads on unreachable regions of the state-space.

The lazy approach. In the lazy approach, we start computing T_1 in the first context from an initial global state. At any point of the execution, we can choose nondeterministically to switch context, in which case we store the value of the shared variables in g_1 and terminate the thread (thereby *losing* its local state).

Then, we execute T_2 in the second context, starting from a local initial state and shared variables initialized with g_1. At any point of the execution, we can choose nondeterministically to switch context, in which case we store in g_2 the value of the shared variables and terminate the thread (again losing the local state).

We then would like to start executing T_1 (i.e., from s_4) in the third context, but since we had lost its local state, we need to recompute it. Thus, starting from an initial global state, we execute T_1 until we reach a state matching g_1 on the shared variables. (Notice that this may be an *entirely* different local state than the one we explored in the first context! However, as we show, this does not affect correctness.)

At such a point, we allow the thread to replace the value of the shared variables with g_2, and proceed to execute T_1 to compute the third context (from s_4 to s_5). Again, at any point, we can decide nondeterministically to switch context, in which case we store the value of the shared variables in g_3, and end thread T_1.

Finally, we move to execute T_2 in the fourth context. As we did for T_1, we need to restart thread T_2 from its initial state to recompute the local state at the beginning of the fourth context (i.e. at s_3 or s_6). Thus, we simulate T_2 with shared variables initialized to g_1, wait for it to reach a state with shared variables matching g_2, non-deterministically choose to assign the global variables with g_3, and proceed to compute the fourth context (from s_6 to s_7). Again, the local state produced on this new invocation of thread T_2 may be entirely different, and yet the states discovered in the fourth context are indeed reachable.

In contrast to the eager approach, the values of g_1, g_2, g_3 are computed dynamically, and thus are guaranteed to be reachable by the concurrent program. This can be sometimes a huge advantage as large (and complex) portions of the state space are spared from analysis. More importantly, we believe that this is a more faithful representation of the concurrent program. On the other hand, it is true that this approach executes each context several times (at most $k/2$ times).

Generalization to multiple threads. Consider now a concurrent program with n threads $T_1, \ldots, T_n$ and a fixed integer $k > 0$. Let the variables g_i $(i = 1, \ldots, k)$ hold the valuation of the shared variables at the i-th context-switch, and let the variables t_i $(i = 0, \ldots, k)$ hold the index of the thread that has control in the $(i+1)$-th context.

In the eager approach, we start guessing non-deterministically both tuples $g_1, \ldots, g_k$ and $t_0, \ldots, t_k$. Then, we run each thread T_i, for $i = 1, \ldots, n$, through all the contexts where it has the control (i.e., all contexts j such that $t_j = i$) and in doing this we check that the context-switches happen at states that match the values of the shared variables in the tuple $g_1, \ldots, g_k$.

In the lazy approach, the thread scheduling $t_0, \ldots, t_k$ is determined non-deterministically as and when the context-switches happen, and each g_i is computed by executing the thread T_j which has the control in the context i (i.e., such that $j = t_i$) through all the contexts in which it had control up until context i. As in the previous approach, when re-executing the thread in the previously computed contexts, we check that the context-switches happen at states that match the values of the shared variables in the already computed tuple $g_1, \ldots, g_{i-1}$.

3 Concurrent Programs

Sequential recursive programs. Let us fix the syntax of a simple *sequential* programming language with variables ranging over only the integer and Boolean

domains, and with explicit syntax for nondeterminism, (recursive) function calls, and tuples of return values.

The transformations in this paper require that we can cache the shared variables, and copy and compare them, and the notation is far simpler when we do not have complex or dynamically allocated structures. Handling domains of types other than integers is straightforward, but we will stick to integers for simplicity.

Programs are described by the following grammar:

$$
\begin{array}{lll}
\langle \textit{pgm} \rangle & ::= & \langle \textit{gvar-decl} \rangle \texttt{;}\ \langle \textit{proc-list} \rangle \\
\langle \textit{gvar-decl} \rangle & ::= & \texttt{decl int}\ x \mid \texttt{decl bool}\ x \mid \langle \textit{gvar-decl} \rangle \texttt{;}\ \langle \textit{gvar-decl} \rangle \\
\langle \textit{proc-list} \rangle & ::= & \langle \textit{proc} \rangle\ \langle \textit{proc-list} \rangle \mid \langle \textit{proc} \rangle \\
\langle \textit{proc} \rangle & ::= & f^{h,m}(x_1, \ldots, x_h)\ \ \texttt{begin}\ \langle \textit{lvar-decl} \rangle \texttt{;}\ \langle \textit{stmt} \rangle\ \texttt{end} \\
\langle \textit{lvar-decl} \rangle & ::= & \texttt{decl int}\ x \mid \texttt{decl bool}\ x \mid \langle \textit{lvar-decl} \rangle \texttt{;}\ \langle \textit{lvar-decl} \rangle \\
\langle \textit{stmt} \rangle & ::= & \langle \textit{stmt} \rangle \texttt{;}\ \langle \textit{stmt} \rangle \mid \texttt{skip} \mid \langle \textit{assign} \rangle \mid \texttt{assume}(\langle \textit{b-expr} \rangle) \mid \\
 & & \texttt{call}\ f^{h,0}(x_1, \ldots, x_h) \ \mid\ \texttt{return}\ x_1, \ldots, x_m \mid \\
 & & \texttt{if}\ (\langle \textit{b-expr} \rangle)\ \texttt{then}\ \langle \textit{stmt} \rangle\ \texttt{else}\ \langle \textit{stmt} \rangle\ \texttt{fi} \mid \\
 & & \texttt{while}\ (\langle \textit{b-expr} \rangle)\ \texttt{do}\ \langle \textit{stmt} \rangle\ \texttt{od} \\
\langle \textit{assign} \rangle & ::= & x_1, \ldots, x_m\ \texttt{:=}\ \langle \textit{expr} \rangle_1, \ldots, \langle \textit{expr} \rangle_m \mid \\
 & & x_1, \ldots, x_m\ \texttt{:=}\ f^{h,m}(y_1, \ldots, y_h) \\
\langle \textit{expr} \rangle & ::= & x \mid c \mid \langle \textit{b-expr} \rangle \\
\langle \textit{b-expr} \rangle & ::= & T \mid F \mid * \mid x \mid \neg \langle \textit{b-expr} \rangle \mid \langle \textit{b-expr} \rangle \vee \langle \textit{b-expr} \rangle
\end{array}
$$

In the above, x, x_i, y_i are from a set of variable names Var, c is any integer constant, and $f^{h,m}$ denotes a function with h formal parameters and m return values. Some of the functions $f^{h,m}$ may be interpreted to have existing functionality, such as integer addition or library functions, in which case their code is not given and we assume they happen atomically.

A program has a global variable declaration followed by a list of functions. Each function is a declaration of local variables followed by a sequence of statements, where statements can be tuple assignments, calls to functions (call-by-value) that take in multiple parameters and return multiple values, conditional statements, while-loops, or return statements. Expressions can be integer constants, variables or Boolean expressions. Boolean expressions can be true, false, or non-deterministically true or false ($*$), and can be combined using standard Boolean operations. Functions that do not return any values are called using the `call` statement. We also assume that the program type-checks with respect to the integer and Boolean types.

We will assume several obvious restrictions on the above syntax: global variables and local variables are assumed to be disjoint; formal parameters are local variables; the body of a function $f^{h,m}$ has only variables that are either globally declared, locally declared, or a formal parameter; a return statement in the body of $f^{h,m}$ is of the form `return` $x_1, \ldots, x_m$ or simply `return` (in the latter case arbitrary values will be returned).

Let us also assume that there is a function `main`, which is the function where the program starts, and that there are no calls to this function in the code of P.

The semantics is the obvious one: a configuration of a program consists of a *stack* which stores the history of positions at which calls were made, along with valuations for local variables, and the top of the stack contains the local and global valuations, and a pointer to the current statement being executed.

The *reachability problem* asks whether a particular statement in the program marked using a special label *Target* is reachable.

Concurrent programs. A *concurrent program* is a finite set of recursive programs running in parallel and sharing some (global) variables.

Formally, the syntax of concurrent programs is defined by extending the syntax of sequential recursive programs with the following rules:

$$\begin{array}{lll} \langle \textit{conc-pgm} \rangle & ::= & \langle \textit{svar-decl} \rangle \texttt{;}\ \langle \textit{init} \rangle\ \langle \textit{pgm-list} \rangle \\ \langle \textit{svar-decl} \rangle & ::= & \texttt{decl int}\ x \mid \texttt{decl bool}\ x \mid \langle \textit{svar-decl} \rangle \texttt{;}\ \langle \textit{svar-decl} \rangle \\ \langle \textit{init} \rangle & ::= & \langle \textit{proc} \rangle \\ \langle \textit{pgm-list} \rangle & ::= & \langle \textit{pgm} \rangle\ \langle \textit{pgm-list} \rangle \mid \langle \textit{pgm} \rangle \end{array}$$

Let $\mathcal{P}$ be a concurrent program formed by the sequential programs $P_1, \ldots, P_n$ (where $n > 0$). Each program P_i has its own global and local variables, and also has access to variables that are shared with the other component programs. Let us assume that each concurrent program has a function `init` where the shared variables are initialized (corresponding to the *init* construct in the above abstract syntax). Let us further assume that the function `main` of each component program P_i is named $\texttt{thread}_i$.

The behavioral semantics of $\mathcal{P}$ is obtained by interleaving the behaviors of $P_1, \ldots, P_n$. At the beginning of any computation the shared variables are set according to function `init`. At any point of a computation, only one of the programs is *active*. Therefore, a *state* of $\mathcal{P}$ is denoted by a tuple $(i, u_S, \overline{u}_1, \ldots, \overline{u}_n)$ where P_i is the currently active program, u_S is a valuation of the shared variables and $\overline{u}_j$ is a state of P_j for $j = 1, \ldots, n$. From such a state the computation of $\mathcal{P}$ can evolve either according to the local behavior of P_i or by switching to another program P_j, which then becomes the new active program. A maximal consecutive part of a computation visiting only states where the same program P_i is active is called a *context*.

The *reachability problem for concurrent programs* asks whether a particular statement in the program marked using a special label *Target* is reachable. The *reachability problem for concurrent programs under a context-switch bound* k, for $k \geq 1$, asks whether *Target* is reachable within k context-switches.

Example 1. Figure 1 illustrates a simple concurrent program with two component programs starting at $\texttt{thread}_1$ and $\texttt{thread}_2$ respectively, and five shared variables $\textit{test}, x_1, \ldots, x_4$, all over the Boolean domain. Function `init` assigns the initial value to the shared variable *test* and leaves all the others unassigned. Function $\texttt{thread}_1$ simply assigns all variables $x_1, \ldots, x_4$ to false, and then sets *test* to true. Function $\texttt{thread}_2$ halts if *test* is false. Otherwise, it will loop forever by nondeterministically choosing either to swap x_1 and x_2, or to shift the bits $x_1, \ldots, x_4$ circularly. The instruction labeled with *Target* is never reached on

```
decl bool test,x1, x2, x3, x4;

void init( ) begin
  test := F;
  return;
end

void thread1( ) begin
  x1, x2, x3, x4 := F, F, F, F;
  test := T;
  return;
end

void thread2( ) begin
  assume(test);
  while(T) do
    if (*) then
      x1, x2, x3, x4 := x2, x1, x4, x3;
    else
      x1, x2, x3, x4 := x2, x3, x4, x1;
    fi
  od
  Target: skip;
  return;
end
```

Fig. 1. The concurrent program *permutation*

any computation. $\texttt{thread}_2$ eventually computes (on various runs) all the permutations of the bits stored in $x_1, \ldots, x_4$. Note that, since these variables are set to false by $\texttt{thread}_1$, the reachable state-space at the while-loop has only the valuation $\langle x_1 = F, x_2 = F, x_3 = F, x_4 = F \rangle$. □

4 Translation Scheme: The Eager Approach

In this section, we give a detailed description of the translation of a concurrent program $\mathcal{P}$ with target program counter *pc* under a context-switching bound k to a sequential program using the eager approach (denoted $\text{Eager}_k(\mathcal{P},pc)$), and argue its correctness.

Besides the variables in $\mathcal{P}$, the sequential program $\text{Eager}_k(\mathcal{P},pc)$ will have extra global variable tuples $g_1, \ldots, g_k$ and $t_0, \ldots, t_k$, as described in Section 2. We add also the following control variables: a variable t to keep the index of the active component program, a variable cx to keep the current context number, a Boolean variable *terminate* to interrupt the execution of a program component, and a Boolean variable *goal* which gets set to true when the target *pc* of $\mathcal{P}$ is reached.

The sequential program $\text{Eager}_k(\mathcal{P},pc)$ is composed of: two new functions `main` and *contextSwitch* that are shown in Figure 2, and for every function P of $\mathcal{P}$, a function P^e which is a transformation of P.

In Figure 2, we use $\mathit{nextContext}(cx, t, t_0, \ldots, t_k)$ to denote a function that computes the index of the first context in which P_t is active after cx, if any, and $k+1$, otherwise. Formally, $\mathit{nextContext}(cx, t, t_0, \ldots, t_k)$ is the value i such that either $t_i = t$ and $t_j \neq t$ for all j s.t. $cx < j < i$, or $i = k+1$ and $t_j \neq t$ for all $j > cx$. Clearly, such an index can be computed with a few lines of code. We also use $\mathit{firstContext}(t, t_0, \ldots, t_k)$ to compute the first context in which P_t is active in the computation.

Let g: shared variables of $\mathcal{P}$; *Let $g_1, \ldots, g_k$ be k copies of g;*
`decl int` $cx, t, t_0, \ldots, t_k$`;` `decl bool` $goal, terminate$`;`

```
void main()
begin
  goal := F;
  assume(⋀_{i=0}^{k} (0 < t_i ≤ n));
  for(i := 1; i ≤ n; i++) do
    t := i;
    cx := firstContext(t, t_0, ..., t_k);
    if (cx ≤ k) then
      if (cx = 0) then init();
      else  g := g_cx; fi
      terminate := F;
      call thread^e_{t_cx}();
    fi
  od
  assume(goal);
  Target: skip;
  return;
end
```

```
void contextSwitch()
begin
  if (cx = k) then
    terminate := T;
  else
    assume(g = g_{cx+1});
    cx := nextContext(cx, t, t_0, ..., t_k);
    if (cx ≤ k) then g := g_cx;
    else terminate := T; fi
  fi
  return;
end
```

Fig. 2. Functions `main` and *contextSwitch* of the program Eager$_k$($\mathcal{P}$,*pc*)

```
if (terminate) then return;
else
     if (∗) then call contextSwitch(); fi
     if (terminate) then return; fi
fi
```

Fig. 3. Control code

Each function P^e is obtained from the corresponding function P of $\mathcal{P}$ by a simple transformation. We first interleave the statements of P with the lines of control code $\mathcal{C}$ shown in Figure 3. More precisely, we rewrite the statements of P according to the following rules ($\mathcal{E}$ is an arbitrary Boolean expression, $\mathcal{S}_1$ and $\mathcal{S}_2$ are arbitrary statements, and $\mathcal{S}$ is a basic statement of the kind `assign` or `skip` or `assume` or `call`):

- $\tau[\mathcal{S}_1; \mathcal{S}_2] = \tau[\mathcal{S}_1]; \tau[\mathcal{S}_2]$
- $\tau[\mathcal{S}] = \mathcal{S}; \mathcal{C}$
- $\tau[$`return` $x_1, \ldots, x_m] =$ `return` $x_1, \ldots, x_m$
- $\tau[$`while` $(\mathcal{E})$ `do` $\mathcal{S}_1$ `od`$] =$ `while` $(\mathcal{E})$ `do` $\mathcal{C}; \tau[\mathcal{S}_1]$ `od`
- $\tau[$`if` $(\mathcal{E})$ `then` $\mathcal{S}_1$ `else` $\mathcal{S}_2$ `fi`$] =$ `if` $(\mathcal{E})$ `then` $\mathcal{C}; \tau[\mathcal{S}_1]$ `else` $\mathcal{C}; \tau[\mathcal{S}_2]$ `fi`

Actually, we can optimize the above translation by inserting control code only after statements that read or write a shared variable, or calls to a function.

Next, we insert "`assume`(F);" before each return statement of the functions `thread`$_i$ for $i = 1, \ldots, n$; this prevents `thread`$_i$ from returning to `main` after executing to completion. Finally, we insert "*goal*:=T;" right before the statement labeled with the target program counter *pc*.

The general behavior of Eager$_k$($\mathcal{P}$,*pc*) is as follows.

Procedure `main` works as a driver program that calls each component program P_i in turn at most once. After checking that $t_0, \ldots, t_n$ all contains valid indices of component programs (i.e., $1 \leq t_i \leq n$ holds for each t_i), for each component program P_i, $i = 1, \ldots, n$, control variables t and cx are assigned respectively with the current component index (i.e., i) and the first context at which the current component is active if any (according to the scheduling of components given by $t_0, \ldots, t_n$). If there is no such context, the next component program is processed. Otherwise, the shared variables are consistently assigned with the initial values (given by `init`, if the current context is the first one, or by g_{cx} otherwise), and $\texttt{thread}_i^e$ is called.

Once called, a component program must run through all the contexts in which it is active and report if the target *pc* is reached by setting *goal* to true. This is ensured by the control code inserted in each function.

Unless the variable *terminate* holds true, after each step of the original program, it is possible (nondeterministically) to call the function *contextSwitch*. In each such call, the computation proceeds by trying to make a context-switch. In case there are no more contexts in which the current program is active, the function sets *terminate* to true and returns. Otherwise, the shared variables are assigned with the guessed values for the next context where the current program is active and *contextSwitch* returns.

Observe that the `if`-statements checking the value of *terminate* in the control code ensure that in case *terminate* is set to true in *contextSwitch*, the control returns from all the calls stored in the call stack up to the `main` function. Since *terminate* is set to true only when there are no more contexts to run for the current component program, it is correct to interrupt the execution of the current component program and return to `main` so that the next component program (if any) can be executed.

Also, observe that the added "`assume`(F);" statement halts any computation that would have reached a return statement in each function $\texttt{thread}_i^e$. Therefore, a call to any $\texttt{thread}_i^e$ returns if and only if all the contexts in which P_i is active have been successfully executed (i.e., *terminate* is set to true).

When all programs have successfully terminated we check the variable *goal*, and if it holds true, then the label *Target* is reached, else the program halts.

Note that if the guessed tuples $g_1, \ldots, g_k$ and $t_0, \ldots, t_k$ do not correspond to an actual computation of $\mathcal{P}$ then there will be at least one component program P_i, with $1 \leq i \leq n$, that cannot match the sequence $g_1, \ldots, g_k$ and thus the call to $\texttt{thread}_i^e$ will not return.

From the above arguments, we conclude that a program counter *pc* is reachable within k context-switches in a computation of $\mathcal{P}$ if and only if *Target* in `main` is reachable in a computation of $\text{Eager}_k(\mathcal{P}, pc)$. Therefore,

Theorem 1. *Given an integer $k \geq 0$, a concurrent program $\mathcal{P}$ and a program counter pc, pc is reachable in $\mathcal{P}$ within at most k context-switches if and only if Target is reachable in $\text{Eager}_k(\mathcal{P}, pc)$.*

5 Translation Scheme: The Lazy Approach

We now detail the description of the translation of a concurrent program $\mathcal{P}$ under a context-switching bound k to a sequential program using the lazy approach (denoted $\text{Lazy}_k(\mathcal{P})$), and argue its correctness.

The structure of $\text{Lazy}_k(\mathcal{P})$ is similar to that of $\text{Eager}_k(\mathcal{P},pc)$. We keep all the variables we used earlier except for t and *goal* which are not needed here. Also, besides variable cx, we need a second global variable, ic, to store a context number (from 1 to k). Recall that for each context c we need to run the component program P_i that is active in c through all the contexts c' in which P_i was active, from the first context through context c. We use cx to store the context c and ic to keep track of the context c' in the above computation.

In $\text{Lazy}_k(\mathcal{P})$ each function P of $\mathcal{P}$ is translated to a function P^l which is similar P^e in Section 4, except that here we do not need to set the global variable *goal* when the target program counter is reached. The remaining functions are `main` and *contextSwitch*, which are significantly different from those described in Section 4, and are given in Figure 4.

For each context cx, function `main` iteratively: (1) chooses the component program P_i which is active in context cx, (2) determines the first context ic in which P_i is active starting from context 0 through context cx, (3) assigns the shared variables consistent with ic (i.e., if $ic = 0$ then the shared variables are assigned by function `init`, otherwise they are assigned with the values they have at the context-switch ic), and (4) calls function $\texttt{thread}_i^l$.

In `main`, the call to function $\texttt{thread}_i^l$ executes the component program P_i^l through all the contexts in which it is active, from the first one up to cx. In fact,

Let g: shared variables of $\mathcal{P}$*;* *Let* $g_1, \ldots, g_k$*: be k copies of g*

```
decl int cx, ic, t_0, ..., t_k;    decl bool terminate;

void main()
begin
  cx := 0;
  while(cx ≤ k) do
    assume(1 ≤ t_cx ≤ n);
    ic := firstContext(t_cx, t_0, ..., t_cx);
    if (ic = 0) then
      init();
    else
      g := g_ic;
    fi
    terminate := F;
    call thread^l_{t_cx}();
    cx := cx + 1;
  od
end

void contextSwitch()
begin
  if (ic = cx) then
    terminate := T;
    if (cx < k) then g_{cx+1} := g; fi
  else
    if (g = g_{ic+1}) then
     ic := nextContext(ic, t_ic, t_0, ..., t_cx);
       g := g_ic;
    fi
  fi
end
```

Fig. 4. Functions `main` and *contextSwitch* of the program $\text{Lazy}_k(\mathcal{P})$

when $\texttt{thread}_i^l$ is called the computation of P_i^l starts at the beginning of context ic (which is set to the first context where P_i^l is active). After any step of P_i^l, it is possible to call *contextSwitch* (*terminate* is set to false before calling $\texttt{thread}_i^l$ in function `main`) and then perform a context-switch if possible.

In function *contextSwitch*, if $ic < cx$, we check if it is possible to perform a context-switch by determining if the current value of the shared variables matches the stored value for the next context-switch. If so, the next context c in which P_i is active is computed and the shared variables are assigned with the value at the beginning of context c (i.e., with g_c). Observe that since we know that P_i is active in cx, there will always be such a context c.

From the last observation, we also have that if $ic < cx$ does not hold then $ic = cx$ must hold. Thus, in the remaining case, i.e. $ic = cx$, it is possible to make a context-switch at any point. This is correct, since we are executing P_i in context cx, which is newly added before making this call to $\texttt{thread}_i^l$ in `main`, and thus we have no requirements to match. When context-switching in this case, we only need to store the values of the shared variables in g_{cx+1} (if we are not yet in the last context k) and flag that this call to $\texttt{thread}_i^l$ has terminated by setting *terminate* to true.

From the above observations, any computation π of $\mathcal{P}$ can be executed step-by-step by program $\text{Lazy}_k(\mathcal{P})$, and thus if π visits a program counter pc, the corresponding computation of $\text{Lazy}_k(\mathcal{P})$ will also visit pc.

Now consider any computation π of $\text{Lazy}_k(\mathcal{P})$ which successfully terminates (i.e., the return statement of function `main` is reached). Recall that by construction each function $\texttt{thread}_i^l$ has a statement "$\texttt{assume}(F)$" guarding each return statement from the original function $\texttt{thread}_i$. Thus, as in the case of the eager approach construction, the only way for a call to $\texttt{thread}_i^l$ to return is by setting the variable *terminate* to true.

From all the above observations, all calls to each $\texttt{thread}_i^l$ made from `main` return if and only if the computed sequence $g_1, \ldots, g_k$ is matched by a computation of $\mathcal{P}$. Therefore, for each computation π of $\text{Lazy}_k(\mathcal{P})$ which successfully terminates and which visits a program counter pc of $\mathcal{P}$ there is a corresponding computation π' of $\mathcal{P}$ which visits pc. Hence,

Theorem 2. *Given an integer $k \geq 0$, a concurrent program $\mathcal{P}$ and a program counter pc, pc is reachable in $\mathcal{P}$ within at most k context-switches if and only if pc is reachable in $\text{Lazy}_k(\mathcal{P})$.*

6 Experiments

Boolean programs. Boolean programs are programs where the data-domain is only Boolean. Since the data-domain is finite, Boolean programs can be subject to *decidable model-checking* for analyzing reachability. In particular, it is well known that sequential Boolean programs have a decidable reachability problem, and that for concurrent Boolean programs, reachability is undecidable (due to multiple stacks). However, it has also been shown that reachability problem for concurrent Boolean programs under a context-switching bound is decidable.

Context switches	1-adder, 1-stopper			2-adders, 1-stopper			1-adder, 2-stoppers			2-adders, 2-stoppers		
		Eager	Lazy		Eager	Lazy		Eager	Lazy		Eager	Lazy
1	N	0.1	0.1	N	0.2	0.1	N	0.1	0.1	N	0.2	0.1
2	N	0.3	0.2	N	0.9	0.8	N	0.7	0.9	N	1.6	2.0
3	N	43.3	1.4	N	135.9	6.3	Y	70.1	0.4	Y	177.6	0.8
4	N	73.6	5.5	Y	1601.0	2.6	Y	597.2	2.9	Y	out of mem.	7.5
5	N	930.0	20.2	Y	-	18.0	Y	-	14.0	Y	out of mem.	66.5
6	N	-	66.8	Y	-	122.9	Y	-	66.1	Y	out of mem.	535.9

Fig. 5. Experimental results on the Bluetooth driver example. Times are expressed in seconds, "-" denotes timeout in 30 minutes and Y/N denote if the target is reachable.

Translation for Boolean programs. We have implemented both the eager and lazy translations described in the previous sections for Boolean programs, in order to reduce the reachability of concurrent Boolean programs under a context-switching bound to the (decidable) problem of sequential Boolean program reachability. We then subject the sequential program to the reachability model-checker Moped [2]. Our goal is to show the applicability of the translations, and in showing that, the lazy translation can outperform the eager one; we show this on a class of Bluetooth driver examples.

The first concurrent program we consider illustrates the difference between the eager and the lazy analysis; we consider the *permutation* example with 16-bits (a four-bit version of this program is shown in Figure 1). Initially, only $\texttt{thread}_1$ can evolve as $\texttt{thread}_2$ is blocked on its first statement. $\texttt{thread}_1$ sets all 16-bits to false and, before returning, sets *test* to true. Now, $\texttt{thread}_2$ can take over and goes into an infinite loop in which the bits are shuffled producing sooner or later all permutations of the 16-bits. Since all the bits are set to false, only one permutation of the 16-bits is possible in the execution. The table with the experiments for *permutation* is reported in Table 1. As it is evident, the eager version simply fails after a few context-switches. In the eager approach all the global variables are guessed at the beginning and all the threads are executed on the guesses. Therefore this does not prevent $\texttt{thread}_2$ to work on arbitrary values of $x_1, \ldots, x_{16}$ and hence ends up exploring a large and complex state-space (storing all permutations requires large BDDs). The lazy version on the other hand only considers reachable states and hence works well on this example.

Context switches	*permutation 16-bits*		
		Eager	Lazy
1	N	6.97	0.1
2	N	194.7	1.0
3	N	out of mem.	122.5

Table 1. Experiments of *permutation* program with 16 bits. Times are expressed in seconds

The second set of experiments is related to a concurrent boolean program modeling a Windows NT Bluetooth driver [11]. Figure 5 depicts the experimental results. This driver has two types of threads: stoppers and adders. A stopper calls a stopping 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, and an error state is reached otherwise. The pending I/O requests to

the driver are maintained in a counter which is modeled with five bits in our Boolean model of the program.

We have considered four different configurations, by considering one or two adders, and one or two stoppers. We translated the corresponding concurrent programs both with the lazy and eager version of our translator allowing up to six context-switches, and run Moped on them. As from Figure 5, the lazy scheme performs a lot better than the eager one, and the performance gap between them increases with the increase in the number of context-switches.

7 Future Directions

There are two interesting future directions we see. One is to see whether using the lazy translation scheme presented here in the context of deductive verification of concurrent C programs as done in [5] leads to more efficient analysis. Second, it would be interesting to see whether the scheme proposed here can be extended to more general concurrent programs with dynamic and unbounded thread creation.

References

1. Bouajjani, A., Fratani, S., Qadeer, S.: Context-bounded analysis of multithreaded programs with dynamic linked structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 207–220. Springer, Heidelberg (2007)
2. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 324–336. Springer, Heidelberg (2001)
3. La Torre, S., Madhusudan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 299–314. Springer, Heidelberg (2008)
4. La Torre, S., Madhusudan, P., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: PLDI (2009)
5. Lahiri, S.K., Qadeer, S., Rakamarić, Z.: Static and Precise Detection of Concurrency Errors in Systems Code using SMT solvers. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 509–524. Springer, Heidelberg (2009)
6. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
7. Lal, A., Touili, T., Kidd, N., Reps, T.W.: Interprocedural analysis of concurrent programs under a context bound. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 282–298. Springer, Heidelberg (2008)
8. Musuvathi, M., Qadeer, S.: Chess: Systematic stress testing of concurrent software. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 15–16. Springer, Heidelberg (2007)
9. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455. ACM, New York (2007)
10. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)

11. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: PLDI, pp. 14–24. ACM, New York (2004)
12. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000)
13. Suwimonteerabuth, D., Esparza, J., Schwoon, S.: Symbolic context-bounded analysis of multithreaded java programs. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 270–287. Springer, Heidelberg (2008)