Context-Bounded Model Checking of Concurrent Software

Shaz Qadeer Jakob Rehof
{qadeer,rehof}@microsoft.com

Microsoft Research

Abstract. The interaction among concurrently executing threads of a program results in insidious programming errors that are difficult to reproduce and fix. Unfortunately, the problem of verifying a concurrent boolean program is undecidable [24]. In this paper, we prove that the problem is decidable, even in the presence of unbounded parallelism, if the analysis is restricted to executions in which the number of context switches is bounded by an arbitrary constant. Restricting the analysis to executions with a bounded number of context switches is unsound. However, the analysis can still discover intricate bugs and is sound up to the bound since within each context, a thread is fully explored for unbounded stack depth. We present an analysis of a real concurrent system by the ZING model checker which demonstrates that the ability to model check with arbitrary but fixed context bound in the presence of unbounded parallelism is valuable in practice. Implementing context-bounded model checking in ZING is left for future work.

1 Introduction

The design of concurrent programs is difficult due to interaction between concurrently executing threads, leading to programming errors that are difficult to reproduce and fix. Therefore, analysis techniques that can automatically detect errors in concurrent programs can be invaluable. In this paper, we present a novel interprocedural static analysis based on model checking [8, 23] for finding subtle safety errors in concurrent programs with unbounded parallelism.

Algorithms exist for checking assertions in a single-threaded boolean program with procedures (and consequently an unbounded stack) [28, 25] and form the basis of a number of efficient static analysis tools [4, 10] for sequential programs. But the same problem is undecidable for multi-threaded programs [24]. As a result, most previous analyses for concurrent programs have suffered from two limitations. Some restrict the synchronization model, which makes the analysis inapplicable to most common concurrent software applications. Other analyses are imprecise either because they are flow-insensitive or because they use decidable but coarse abstractions. This limitation makes it extremely difficult to report errors accurately to programmers. As a result, these analyses have seen limited use in checking tools for concurrent software. We present a more detailed discussion of related work in Section 6.

In this paper, we take a different approach and focus on the following decision problem:

> Given a multithreaded boolean program P and a positive integer k, does P go wrong by failing an assertion via an execution with at most k contexts?

A *context* is an uninterrupted sequence of actions by a *single* thread. Thus, in an execution with k contexts execution switches from one thread to another $k-1$ times. We prove that this problem is decidable and present an algorithm that is polynomial in the size of P and exponential in the parameter k.

Our technique, although unsound in general, is both sound and precise for context-bounded executions of concurrent programs. We believe that it can catch nontrivial safety errors caused by concurrency. First, even though our analysis bounds the number of contexts in an execution, it fully explores a thread within each context. Due to recursion within a thread, the number of stack configurations explored within a context is potentially unbounded. Our analysis considers each such reachable configuration as a potential point for a context switch and schedules all other threads from it. Second, our experience analyzing low-level systems code with the KISS checker [22] indicates that a variety of subtle bugs caused by concurrency are manifested by executions with few contexts.

Our work is inspired by the KISS project but significantly extends its scope by employing entirely different techniques. The KISS checker simulates executions of a concurrent program P with the executions of a sequential program P' derived from P. The various threads in P are scheduled using the single stack of P'. The use of a single stack fundamentally limits the number of context switches that can be explored. KISS is unable to explore more than two context switches for a concurrent program with two threads and cannot handle an unbounded number of threads. This paper presents a general algorithm for exploring an arbitrary number of context switches, even in the presence of unbounded parallelism, in a way that is sound and precise up to the bound.

The main difficulty with context-bounded model checking is that in each thread context, an unbounded number of stack configurations could be reachable due to recursion. Since a context switch may happen at any time, a precise analysis must schedule other threads from each of these configurations. To guarantee termination, a systematic state exploration algorithm must use a finite representation of an unbounded set of stack configurations. Our previous algorithm based on transactions and procedure summaries [20] is not guaranteed to terminate for context-bounded model checking because it keeps an explicit representation of the stack of each thread. Summarization [20] may still be useful as an optimization technique that is complementary to the techniques presented in this paper.

We achieve a finite representation of an unbounded set of stack configurations by appealing to the result that the reachable configurations (sometimes called the pushdown store language) of a pushdown system is regular [3, 12, 27] and consequently representable by a finite automaton. We use this fact to design an algorithm for context-bounded model checking for a concurrent boolean program with a finite but arbitrary number of threads. We then consider the main problem of this paper, context-bounded model checking of dynamic concurrent boolean programs. A dynamic concurrent boolean program is allowed to use two new operators. The *fork* operator creates a new thread and returns an integer identifying the new thread. The *join* operator blocks until the thread identified by an argument to the operation has terminated. We assume that *fork*, *join*, and copy from one variable to another are the only operations on thread identifiers. We show that for any context bound k and for any dynamic concurrent boolean program P, we can construct a concurrent boolean program Q with $k+1$ threads such that it suffices to check Q rather than P. Since concurrent software invariably uses dynamic thread creation, this result significantly increases the applicability of context-bounded model checking.

Proofs of the theorems in this paper can be found in our report [21].

2 Example

```
void Remove( LtmInternalTransaction tx )
{
  if( !tx.inTimerList )
  {
    // This transaction is not in the list.
    return;
  }
POINT 1:
  lock( this ) // lock bucket of hash table
  {
    if( tx.nextLink != null )
    {
POINT 3:
      tx.nextLink.prevLink = tx.prevLink;
    }
    if( tx.prevLink != null )
    {
      // ERROR: null pointer dereference
      tx.prevLink.nextLink = tx.nextLink;
    }
    ...
}
```

```
bool ProcessList()
{
  LtmInternalTransaction tx;
  long expirationTime = DateTime.UtcNow.Ticks;

  do
  {
    tx = null;
    lock( this )  // lock bucket of hash table
    {
     ... // remove transaction from timeout list
     ... // tx is made non-null here
    }
    if( tx != null )
    {
      tx.prevLink = null;
POINT 2:
      tx.nextLink = null;
POINT 4:
      ...
    }
  } while( tx != null );
}
```

Fig. 1. Example

In this section, we present an example of a real concurrency error that requires four context switches to manifest itself. The error was found by model checking a large transaction management system written in C# with a bounded number of threads, using the model checker ZING [2] after compiling 10,000 lines of C# code into ZING. Since the error cannot manifest itself with fewer than four context switches, it could not be discovered by the techniques of KISS [22] which are inherently limited to two context switches.

The code shown in Figure 1 contains excerpts from two methods of a hashtable class that is part of the transaction manager implementing the two-phase commit protocol. The methods `Remove` and `ProcessList` are found within a class that implements a bucket of the hashtable. When a client thread is registered with the transaction manager, a reference to the thread is stored in the hashtable. The error arises when a thread T_c has comitted a transaction `tx` and executes in the `Remove` method in order to remove the finished transaction from the appropriate bucket of the hashtable. At the same time, a timer thread T_t is executing in the `ProcessList` method to determine if any of the transactions referenced in the bucket of the hashtable has timed out. Thread T_c gets interrupted at `POINT 1` in the `Remove` method, just after it has tested that `tx` has not already been removed by the timer and just before it tries to acquire a lock on the bucket in order to remove the transaction. Thread T_t acquires the lock on the same bucket inside `ProcessList`, and it decides that transaction `tx` has timed out. It goes on to remove `tx` by setting the bucket links in `tx` to `null` (the bucket is a doubly-linked list). Just before setting `tx.nextLink` to `null`, another context switch occurs, at `POINT 2`. Thread T_c resumes execution at `POINT 1` and learns that `tx.nextLink` is non-`null`. It gets interrupted by thread T_t at `POINT 3` which resumes execution at `POINT 2` and sets `tx.nextLink` to `null`. It gets interrupted by thread T_c at `POINT 4`. Thread T_c resumes execution at `POINT 3` and dereferences the `null` pointer `tx.nextLink`. The error can be fixed by extending the scope of the `lock` statement in the `ProcessList` method down to `POINT 4`.

We have been able to discover several other bugs in the system of the same nature. However, we have not been able to check the system under scenarios in which asynchronous calls and dynamically created timers may create new threads, because ZING may not terminate on programs with unbounded parallelism. The results of this paper show that we can achieve a finite abstraction by bounding the number of contexts to an arbitrary constant, even in the presence of dynamic thread creation. It is an important problem for future work to integrate our algorithm in ZING, thereby enabling us to find deep errors such as the one shown above, even in the presence of unbounded parallelism.

3 Pushdown systems

Domains

$\gamma \in \Gamma$	*Stack alphabet*
$w \in \Gamma^*$	*Stack*
$g \in G$	*Global state*
$\Delta \subseteq (G \times \Gamma) \times (G \times \Gamma^*)$	*Transition relation*
$c \in G \times \Gamma^*$	*Configuration*
$\longrightarrow_\Delta \subseteq (G \times \Gamma^*) \times (G \times \Gamma^*)$	*Pds transition*

Let G and Γ be arbitrary fixed finite sets. We refer to G as the set of *global states*, and we refer to Γ as the *stack alphabet*. We let g range over elements of

G, and we let γ range over elements of Γ. A *stack* w is an element of Γ^*, the set of finite strings over Γ, including the empty string ϵ. A *configuration* c is an element of $G \times \Gamma^*$; we write configurations as $c = \langle g, w \rangle$ with $g \in G$ and $w \in \Gamma^*$.

A *transition relation* Δ over G and Γ is a finite subset of $(G \times \Gamma) \times (G \times \Gamma^*)$. A *pushdown system* $P = (G, \Gamma, \Delta, g_{in}, w_{in})$ is given by G, Γ, a transition relation Δ over G and Γ, and an initial configuration $\langle g_{in}, w_{in} \rangle$. The transition relation Δ determines a transition system on configurations, denoted $\longrightarrow_\Delta$, as follows: $\langle g, \gamma w' \rangle \longrightarrow_\Delta \langle g', w w' \rangle$ for all $w' \in \Gamma^*$, if and only if $(\langle g, \gamma \rangle, \langle g', w \rangle) \in \Delta$. We write $\longrightarrow_\Delta^*$ to denote the reflexive, transitive closure of $\longrightarrow_\Delta$. Notice that, by the signature of Δ, there are no transitions $\longrightarrow_\Delta$ from a configuration whose stack is empty. Hence, a pushdown system as defined here halts when the stack becomes empty.

A configuration c of a pushdown system is called *reachable* if and only if $c_{in} \longrightarrow_\Delta^* c$, where c_{in} is the initial configuration of the pushdown system. In general, there are inifinitely many reachable configurations of a pushdown system, because the stack is unbounded.

The reachability problem for pushdown systems is decidable because the set of reachable configurations (sometimes called the pushdown store language) of a pushdown system is regular [3,12]. A *regular pushdown store automaton* $A = (Q, \Gamma, \delta, I, F)$ is a finite automaton with states Q, alphabet Γ, transition relation $\delta \subseteq Q \times \Gamma \times Q$, initial states I and final states F. The automaton may contain ϵ-transitions. The sets Q and I satisfy $G \subseteq Q$ and $I \subseteq G$. Such an automaton defines a language of pushdown configurations by the rule [27]:

- A accepts a pushdown configuration $\langle g, w \rangle$, if and only if A accepts the word w when started in the state g.

A subset $S \subseteq G \times \Gamma^*$ of pushdown configurations is called *regular*, if and only if there exists a regular pushdown store automaton A such that $S = L(A)$.

For a pushdown system $P = (G, \Gamma, \Delta, g_{in}, w_{in})$ and a set of configurations $S \subseteq G \times \Gamma^*$, let $Post_\Delta^*(S)$ be the set of configurations reachable from S, i.e., $Post_\Delta^*(S) = \{c \mid \exists c' \in S.\ c' \longrightarrow_\Delta^* c\}$. The following theorem [27] shows that the set of reachable configurations from a regular set of configurations is again regular. For details on the construction leading to this result we refer the reader to [27].

Theorem 1 ([27]). *Let $P = (G, \Gamma, \Delta, g_{in}, w_{in})$ be a pushdown system, and let A be a regular pushdown store automaton. There exists a regular pushdown store automaton A' such that $Post_\Delta^*(L(A)) = L(A')$. The automaton A' can be constructed from P and A in time polynomial in the size of P and A.*

4 Concurrent pushdown systems

A *concurrent pushdown system* is a tuple $P = (G, \Gamma, \Delta_0, \ldots, \Delta_N, g_{in}, w_{in})$ with transition relations $\Delta_0, \ldots, \Delta_N$ over G and Γ, $N \geq 0$, an initial state g_{in} and an initial stack w_{in}. A configuration of a concurrent pushdown system is a tuple

$c = \langle g, w_0, \ldots, w_N \rangle$ with $g \in G$ and $w_i \in \Gamma^*$, that is, a global state g followed by a sequence of stacks w_i, one for each constituent transition relation. The initial configuration of P is $\langle g_{in}, w_{in}, \ldots, w_{in} \rangle$ where all $N+1$ stacks are initialized to w_{in}. The transition system of P, denoted $\longrightarrow_P$, rewrites configurations of P by rewriting the global state together with any one of the stacks, according to the transition relations of the constituent pushdown systems. Formally, we define $\langle g, w_0, \ldots, w_i, \ldots w_N \rangle \longrightarrow_i \langle g', w_0, \ldots, w_i', \ldots w_N \rangle$ if and only if $\langle g, w_i \rangle \longrightarrow_{\Delta_i} \langle g', w_i' \rangle$. We define the transition relation $\longrightarrow_P$ on configurations of P by the union of the $\longrightarrow_i$, i.e., $\longrightarrow_P = \bigcup_{i=0}^{N} \longrightarrow_i$.

4.1 Bounded reachability

A configuration c is called reachable, if and only if $c_{in} \longrightarrow_P^* c$, where c_{in} is the initial configuration. The reachability problem for concurrent pushdown systems is undecidable [24]. However, as we will show below, bounding the number of context switches allowed in a transition leads to a decidable restriction of the reachability problem.

For a positive natural number k, we define the *k-bounded transition relation* $\xrightarrow{k}$ on configurations c inductively, as follows:

$$
\begin{array}{l}
c \xrightarrow{1} c' \text{ iff there exists } i \text{ such that } c \longrightarrow_i^* c' \\
c \xrightarrow{k+1} c' \text{ iff there exist } c'' \text{ and } i \text{ such that } c \xrightarrow{k} c'' \text{ and } c'' \longrightarrow_i^* c'
\end{array}
$$

Thus, a k-bounded transition contains at most $k-1$ "context switches" in which a new relation $\longrightarrow_i$ can be chosen. Notice that the full transitive closure of each transition relation $\longrightarrow_i$ is applied within each context. We say that a configuration c is *k-reachable* if $c_{in} \xrightarrow{k} c$. The *k-bounded reachability problem* for a concurrent pushdown system P is: *Given configurations* c_0 *and* c_1*, is it the case that* $c_0 \xrightarrow{k} c_1$*?*

For fixed k, the lengths and state spaces of k-bounded transition sequences may be unbounded, since each constituent transition relation $\longrightarrow_i^*$ may generate infinitely many transitions containing infinitely many distinct configurations. Therefore, decidability of k-bounded reachability requires an argument. In order to formulate this argument, we will define a transition relation over *aggregate configurations* of the form $\langle\langle g, R_0, \ldots, R_N \rangle\rangle$, where R_i are regular subsets of Γ^*.

For a global state $g \in G$ and a regular subset $R \subseteq \Gamma^*$, we let $\langle\langle g, R \rangle\rangle$ denote the set of configurations $\{\langle g, w \rangle \mid w \in R\}$. Notice that $\langle\langle g, \emptyset \rangle\rangle = \emptyset$. For $G = \{g_1, \ldots, g_m\}$, any regular set of configurations $S \subseteq G \times \Gamma^*$ can evidently be written as a disjoint union: $(*)$ $S = \biguplus_{i=1}^{m} \langle\langle g_i, R_i \rangle\rangle$ for some regular sets of stacks $R_i \subseteq \Gamma^*$, $i = 1 \ldots m$ (if there is no configuration with global state g_j in S, then we take $R_j = \emptyset$.) By Theorem 1, the set $Post_\Delta^*(S)$ for regular S can also be written in the form $(*)$, since it is a regular set. We abuse set membership notation to denote that $\langle\langle g', R' \rangle\rangle$ is a component of the set $Post_\Delta^*(S)$ as represented in the form $(*)$, writing $\langle\langle g', R' \rangle\rangle \in Post_\Delta^*(S)$ if and only if $Post_\Delta^*(S) = \biguplus_{i=1}^{m} \langle\langle g_i, R_i \rangle\rangle$ with $\langle\langle g', R' \rangle\rangle = \langle\langle g_j, R_j \rangle\rangle$ for some $j \in \{1, \ldots, m\}$.

Input: Concurrent pushdown system $(G, \Gamma, \Delta_0, \ldots, \Delta_N, g_{in}, w_{in})$ *and bound* k

let $A_{in} = (Q, \Gamma, \delta, \{g_{in}\}, F)$ such that $L(A_{in}) = \{\langle g_{in}, w_{in}\rangle\}$;

WL := $\{(\langle g, A_{in}, \ldots, A_{in}\rangle, 0)\}$; // There are N copies of A_{in}
$Reach$:= $\{\langle g, A_{in}, \ldots, A_{in}\rangle\}$;

while (WL **not empty**)
  let $(\langle g, A_0, \ldots, A_N\rangle, i) = \text{REMOVE}(WL)$ **in**
    if $(i < k)$
      forall $(j = 0 \ldots N)$
        let $A'_j = Post^*_{\Delta_j}(A_j)$ **in**
          forall $(g' \in G(A'_j))$ {
            let $x = \langle g', \text{RENAME}(A_0, g'), \ldots, \text{ANONYMIZE}(A'_j, g'), \ldots, \text{RENAME}(A_N, g')\rangle$ **in**
              $\text{ADD}(WL, (x, i+1))$;
              $Reach$:= $Reach \cup \{x\}$;
          }
Output : $Reach$

Fig. 2. Algorithm

Given a concurrent pushdown system $P = (G, \Gamma, \Delta_0, \ldots, \Delta_N, g_{in}, w_{in})$, we define relations $\Longrightarrow_i$ on aggregate configurations, for $i = 0 \ldots N$, by $\langle\langle g, R_0, \ldots, R_i, \ldots, R_N\rangle\rangle \Longrightarrow_i \langle\langle g', R_0, \ldots, R'_i, \ldots, R_N\rangle\rangle$ if and only if $\langle\langle g', R'_i\rangle\rangle \in Post^*_{\Delta_i}(\langle\langle g, R_i\rangle\rangle)$. Finally, define the transition relation $\Longrightarrow$ on aggregate configurations by the union of the $\Longrightarrow_i$, i.e., $\Longrightarrow\; = \;(\bigcup_{i=0}^{N} \Longrightarrow_i)$. For aggregate configurations a_1 and a_2, we write $a_1 \overset{k}{\Longrightarrow} a_2$, if and only if there exists a transition sequence using $\Longrightarrow$ starting at a_1 and ending at a_2 with at most k transitions. Notice that each relation $\Longrightarrow_i$ contains the full transitive closure computed by the $Post^*_{\Delta_i}$ operator.

The following theorem reduces k-bounded reachability in a concurrent pushdown system to repeated applications of the sequential $Post^*$ operator.

Theorem 2. *Let* $P = (G, \Gamma, \Delta_0, \ldots, \Delta_N, g_{in}, w_{in})$ *be a concurrent pushdown system. Then, for any* k, *we have* $\langle g, w_0, \ldots, w_N\rangle \xrightarrow{k} \langle g', w'_0, \ldots, w'_N\rangle$ *if and only if* $\langle\langle g, \{w_0\}, \ldots, \{w_N\}\rangle\rangle \overset{k}{\Longrightarrow} \langle\langle g', R'_0, \ldots, R'_N\rangle\rangle$ *for some* $R'_0, \ldots, R'_N$ *such that* $w'_i \in R'_i$ *for all* $i \in \{0, \ldots, N\}$.

4.2 Algorithm

Theorem 1 and Theorem 2 together give rise to an algorithm for solving the context-bounded reachability problem for concurrent pushdown systems. The algorithm is shown in Figure 2.

The algorithm processes a worklist WL containing a set of items of the form $(\langle g, A_0, \ldots, A_N\rangle, i)$, where $g \in G$ is a global state, the A_j are pushdown store automata, and i is an index in the range $\{0, \ldots, k-1\}$. The operation REMOVE(WL) removes an item from the worklist and returns the item; ADD($WL, item$) adds the item to the worklist. The initial pushdown store automaton $A_{in} = (Q, \Gamma, \delta, \{g_{in}\}, F)$ has initial state g_{in} and accepts exactly the initial configuration $\langle g_{in}, w_{in} \rangle$. In the line numbered 7 of the algorithm in Figure 2, the pushdown store automaton $A'_j = Post^*_{\Delta_j}(A_j)$ is understood to be constructed according to Theorem 1 so that $L(A'_j) = Post^*_{\Delta_j}(L(A_j))$. In line 8, $G(A'_j) = \{g' \mid \exists w. \langle g', w \rangle \in L(A'_j)\}$. All pushdown store automata A_j constructed by the algorithm have at most one start state $g \in G$. When applied to such an automaton RENAME(A, g') returns the result of renaming the start state if any of A to g'. The operation ANONYMIZE(A, g') is obtained from A by renaming all states of A except g' to fresh states that are not in G.

The algorithm in Figure 2 works by repeatedly applying the $Post^*$ operator to regular pushdown store automata that represent components in aggregate configurations. The operations RENAME and ANONYMIZE are necessary for applying Theorem 1 repeatedly, since the construction of pushdown store automata [27] uses elements of G as states in these automata. In order to avoid confusion between such states across iterated applications of Theorem 1, renaming on states from G is necessary, and hence successive pushdown store automata constructed by the algorithm in Figure 2 may grow for increasing values of the bound k. This factunderlies the undecidability of the unbounded reachability problem.

Theorem 3. *Let $P = (G, \Gamma, \Delta_0, \ldots, \Delta_N, g_{in}, w_{in})$ be a concurrent pushdown system. For any k, the algorithm in Figure 2 terminates on input P and k, and $\langle\!\langle g_{in}, \{w_{in}\}, \ldots, \{w_{in}\} \rangle\!\rangle \overset{k}{\Longrightarrow} \langle\!\langle g', R'_0, \ldots, R'_N \rangle\!\rangle$ if and only if the algorithm outputs Reach with $\langle g', A'_0, \ldots, A'_N \rangle \in Reach$ such that $L(A'_i) = \langle\!\langle g', R'_i \rangle\!\rangle$ for all $i \in \{0, \ldots, N\}$.*

Theorem 2 together with Theorem 3 imply that the algorithm in Figure 2 solves the context-bounded model checking problem, since Theorem 2 shows that aggregate configurations correctly represent reachability in the relation $\xrightarrow{k}$.

For a concurrent pushdown system $P = (G, \Gamma, \Delta_0, \ldots, \Delta_N, g_{in}, w_{in})$ we measure the size of P by $|P| = \max(|G|, |\Delta_0|, |\Delta_1|, \ldots, |\Delta_N|, |\Gamma|)$. For a pushdown store automaton $A = (Q, \Gamma, \delta, I, F)$ we measure the size of A by $|A| = \max(|Q|, |\delta|, |\Gamma|)$.

Theorem 4. *For a concurrent pushdown system $P = (G, \Gamma, \Delta_0, \ldots, \Delta_N, g_{in}, w_{in})$ and a bound k, the algorithm in Figure 2 decides the k-bounded reachability problem in time $\mathcal{O}(k^3 (N|G|)^k |P|^5)$.*

5 Dynamic concurrent pushdown systems

In this section, we define a dynamic concurrent pushdown system with operations for forking and joining on a thread. To allow for dynamic fork-join parallelism, we allow program variables in which thread identifiers can be stored. Thread identifiers are members of the set $Tid = \{0, 1, 2, \ldots\}$. The identifier of a forked thread may be stored by the parent thread in such a variable. Later, the parent thread may perform a join on the thread identifier contained in that variable.

Formally, a dynamic concurrent pushdown system is a tuple

$$(GBV, GTV, LBV, LTV, \Delta, \Delta_F, \Delta_J, g_{in}, \gamma_{in}).$$

The various components of this tuple are described below.

- GBV is the set of global variables containing boolean values and GTV is the set of global variables containing thread identifiers. Let G be the (infinite) set of all valuations to the global variables.
- LBV is the set of local variables containing boolean values and LTV is the set of local variables containing thread identifiers. Let Γ be the (infinite) set of all valuations to the local variables.
- $\Delta \subseteq (G \times \Gamma) \times (G \times \Gamma^*)$ is the transition relation describing a single step of any thread.
- $\Delta_F \subseteq Tid \times (G \times \Gamma) \times (G \times \Gamma^*)$ is the fork transition relation. If $(t, \langle g, \gamma \rangle, \langle g', w \rangle) \in \Delta_F$, then in the global store g a thread with γ at the top of its stack may fork a thread with identifier t modifying the global store to g' and replacing γ at the top of the stack with w.
- $\Delta_J \subseteq LTV \times (G \times \Gamma) \times (G \times \Gamma^*)$ is the join transition relation. If $(x, \langle g, \gamma \rangle, \langle g', w \rangle) \in \Delta_J$, then in the global store g a thread with γ at the top of its stack blocks until the thread with identifier $\gamma(x)$ finishes execution. On getting unblocked, this thread modifies the global store to g' and replaces γ at the top of the stack with w.
- g_{in} is a fixed valuation to the set of global variables such that $g_{in}(x) = 0$ for all $x \in GTV$.
- γ_{in} is a fixed valuation to the set of local variables such that $\gamma_{in}(x) = 0$ for all $x \in LTV$.

Domains

$$\begin{array}{ll}
ss \in Stacks = Tid \rightarrow (\Gamma \cup \{\$\})^* & \\
c \in C = G \times Tid \times Stacks & \textit{Configuration} \\
\rightsquigarrow \subseteq C \times C &
\end{array}$$

Every dynamic concurrent pushdown system is equipped with a special symbol $\$ \notin \Gamma$ to mark the bottom of the stack of each thread. A configuration of the system is a triple $\langle g, n, ss \rangle$, where g is the global state, n is the identifier of the last thread to be forked, and $ss(t)$ is the stack for thread $t \in Tid$. The execution of the dynamic concurrent pushdown system starts in the configuration $\langle g_{in}, 0, ss_0 \rangle$, where $ss_0(t) = \gamma_{in}\$$ for all $t \in Tid$. The rules shown below define the transitions that may be performed by thread t from a configuration $\langle g, n, ss \rangle$.

Operational semantics

(SEQ)
$$\frac{t \leq n \quad ss(t) = \gamma w' \quad (\langle g, \gamma\rangle, \langle g', w\rangle) \in \Delta}{\langle g, n, ss\rangle \rightsquigarrow_t \langle g', n, ss[t := ww']\rangle}$$

(SEQEND)
$$\frac{t \leq n \quad ss(t) = \$}{\langle g, n, ss\rangle \rightsquigarrow_t \langle g, n, ss[t := \epsilon]\rangle}$$

(FORK)
$$\frac{t \leq n \quad ss(t) = \gamma w' \quad (n+1, \langle g, \gamma\rangle, \langle g', w\rangle) \in \Delta_F}{\langle g, n, ss\rangle \rightsquigarrow_t \langle g', n+1, ss[t := ww']\rangle}$$

(JOIN)
$$\frac{t \leq n \quad ss(t) = \gamma w' \quad x \in LTV \quad (x, \langle g, \gamma\rangle, \langle g', w\rangle) \in \Delta_J \quad ss(\gamma(x)) = \epsilon}{\langle g, n, ss\rangle \rightsquigarrow_t \langle g', n, ss[t := ww']\rangle}$$

All rules are guarded by the condition $t \leq n$ indicating that thread t must have already been forked. Thus, only thread 0 can make a move from the initial configuration $\langle g_{in}, 0, ss_0\rangle$. The rule (SEQ) allows thread t to perform a transition according to the transition relation Δ. The rule (SEQEND) is enabled if the top (and the only) symbol on the stack of thread t is \$. The transition pops the \$ symbol from the stack of thread t without changing the global state so that thread t does not perform any more transitions. The rule (FORK) creates a new thread with idenfier $n+1$. The rule (JOIN) is enabled if thread $\gamma(x)$, where γ is the symbol at the top of the stack of thread t, has terminated. The termination of a thread is indicated by an empty stack.

5.1 Assumptions

In realistic concurrent programs with fork-join parallelism, the usage of thread identifiers (and consequently variables containing thread identifiers) is restricted. A thread identifier is created by a fork operation and stored in a variable. Then, it may be copied from one variable to another. Finally, a join operation may look at a thread identifier contained in such a variable. In a nutshell, no control flow other than that implicit in a join operation depends on thread identifiers. We exploit the restricted use of thread identifiers in concurrent systems to devise an algorithm for solving the k-bounded reachability problem.

To formalize the assumptions about the restricted use of thread identifiers, we need the notion of a renaming function. A renaming function is a partial function from *Tid* to *Tid*. When a renaming function f is applied to a global store g, it returns another store in which the value of each variable of type *Tid* is transformed by an application of f. If f is undefined on the value of some global variable in g, it is also undefined on g. Similarly, a renaming function can be applied to a local store as well. A renaming function is extended to a sequence of local stores by pointwise application to each element of the sequence.

$$\begin{tikzcd}
\langle g, \gamma \rangle \arrow[r, "\Delta"] \arrow[d, "f"] & \langle g', w \rangle \arrow[d, "f"] \\
\langle f(g), f(\gamma) \rangle \arrow[r, "\Delta"] & \langle fg', fw \rangle
\end{tikzcd}$$

Fig. 3. Pictorial view of assumption about Δ.

Figure 3 presents a pictorial view of assumption about Δ. This view shows four arrows, two horizontal labeled with the transition relation Δ and two vertical labeled with the renaming function f. The assumption on Δ expresses two requirement on tuples (g, γ) for which the left vertical arrow holds: (1) If the top horizontal arrow holds in addition, then the remaining two arrows hold. (2) If the bottom horizontal arrow holds in addition, then the remaining two arrows hold. Assumptions about Δ_F and Δ_J are similar in spirit to Δ. For lack of space, we leave the formal statements of these assumptions in our technical report [21].

5.2 Reduction to concurrent pushdown systems

In this section, we show how to reduce the problem of k-bounded reachability on a dynamic concurrent pushdown system to a concurrent pushdown system with $k+1$ threads. Given a dynamic concurrent pushdown system P and a positive integer k, our method produces a concurrent pushdown system P_k containing $k+1$ threads with identifiers in $\{0, 1, \ldots, k\}$ such that it suffices to verify the k-bounded executions of P_k. The latter problem can be solved using the algorithm in Figure 2.

The key insight behind our approach is that in a k-bounded execution, at most k different threads may perform a transition. We would like to simulate transitions of these k threads with transitions of threads in P_k with identifiers in $\{0, \ldots, k-1\}$. The last thread in P_k with identifier k never performs a transition; it exists only to simulate the presence of the remaining threads in P.

Let $Tid_k = \{0, 1, \ldots, k\}$ be the set of the thread identifiers bounded by k. Let $AbsG_k$ and $Abs\Gamma_k$ be the set of all valuations to global and local variables respectively, where the variables containing thread identifiers only take values from Tid_k. Note that both $AbsG_k$ and $Abs\Gamma_k$ are finite sets.

Given a dynamic concurrent pushdown system

$$P = (GBV, GTV, LBV, LTV, \Delta, \Delta_F, \Delta_J, g_{in}, \gamma_{in})$$

and a positive integer k, we define a concurrent pushdown system

$$P_k = (AbsG_k \times Tid_k \times \mathcal{P}(Tid_k), Abs\Gamma_k \cup \{\$\}, \Delta_0, \ldots, \Delta_k, (g_{in}, 0, \emptyset), \gamma_{in}\$).$$

The concurrent pushdown system P_k has $k+1$ threads. A global state of P_k is 3-tuple (g, n, α), where g is a valuation to the global variables, n is the largest thread identifier whose corresponding thread is allowed to make a transition, and α is the set of thread identifiers whose corresponding threads have terminated. The initial global state is $(g_{in}, 0, \emptyset)$, which indicates that initially only thread 0 can perform a transition and no thread has finished execution. The rules below define the transitions in the transition relation Δ_t of thread t.

Definition of Δ_t

(ABSSEQ)
$$\frac{t \leq n \quad (\langle g, \gamma\rangle, \langle g', w\rangle) \in \Delta}{(\langle (g, n, \alpha), \gamma\rangle \ ,\ \langle (g', n, \alpha), w\rangle) \in \Delta_t}$$

(ABSSEQEND)
$$\frac{t \leq n}{(\langle (g, n, \alpha), \$\rangle \ ,\ \langle (g, n, \alpha \cup \{t\}), \epsilon\rangle) \in \Delta_t}$$

(ABSFORK)
$$\frac{t \leq n \quad n+1 < k \quad (n+1, \langle g, \gamma\rangle, \langle g', w\rangle) \in \Delta_F}{(\langle (g, n, \alpha), \gamma\rangle \ ,\ \langle (g', n+1, \alpha), w\rangle) \in \Delta_t}$$

(ABSFORKNONDET)
$$\frac{t \leq n \quad (k, \langle g, \gamma\rangle, \langle g', w\rangle) \in \Delta_F}{(\langle (g, n, \alpha), \gamma\rangle \ ,\ \langle (g', n, \alpha), w\rangle) \in \Delta_t}$$

(ABSJOIN)
$$\frac{t \leq n \quad x \in LTV \quad (x, \langle g, \gamma\rangle, \langle g', w\rangle) \in \Delta_J \quad \gamma(x) \in \alpha}{(\langle (g, n, \alpha), \gamma\rangle \ ,\ \langle (g', n, \alpha), w\rangle) \in \Delta_t}$$

Note that all rules above are guarded by the condition $t \leq n$ to indicate that no transition in thread t is enabled in $\langle (g, n, \alpha), \gamma\rangle$ if $t > n$. The rule (ABSSEQ) adds transitions in Δ to Δ_t. The rule (ABSSEQEND) adds thread t to the set of terminated threads. The rules (ABSFORK) and (ABSFORKNONDET) handle thread creation in P and are the most crucial part of our transformation. The rule (ABSFORK) handles the case when the new thread being forked participates in a k-bounded execution. This rule increments the counter n allowing thread $n+1$ to begin simulating the newly forked thread. The rule (ABSFORKNONDET) handles the case when the new thread being forked does not pariticipate in a k-bounded execution. This rule leaves the counter n unchanged thus conserving the precious resource of thread identifiers in P_k. Both these rules add the transitions of the forking thread in Δ_F to Δ. The rule (ABSJOIN) handles the join operator by using the fact that the identifiers of all previously terminated threads are present in α. Again, this rule adds the transitions of the joining thread in Δ_J to Δ.

We can now state the correctness theorems for our transformation. To simplify the notation required to state these theorems, we write a configuration $\langle (g', n', \alpha), w_0, w_1, \ldots, w_k\rangle$ of P_k as $\langle (g', n', \alpha), ss'\rangle$, where ss' is a map from Tid_k to $(Abs\Gamma_k \cup \$)^*$.

First, our transformation is sound which means that by verifying P_k, we do not miss erroneous k-bounded executions of P.

Theorem 5 (Soundness). *Let P be a dynamic concurrent pushdown system and k be a positive integer. Let $\langle g, n, ss\rangle$ be a k-reachable configuration of P. Then there is a total renaming function $f : Tid \rightarrow Tid_k$ and a k-reachable configuration $\langle (g', n', \alpha), ss'\rangle$ of the concurrent pushdown system P_k such that $g' = f(g)$ and $ss'(f(j)) = f(ss(j))$ for all $j \in Tid$.*

Second, our transformation is precise which means that every erroneous k-bounded execution of P_k corresponds to an erroneous execution of P.

Theorem 6 (Completeness). *Let P be a dynamic concurrent pushdown system and k be a positive integer. Let $\langle (g', n', \alpha), ss' \rangle$ be a k-reachable configuration of the concurrent pushdown system P_k. Then there is a total renaming function $f : Tid \rightarrow Tid_k$ and a k-reachable configuration $\langle g, n, ss \rangle$ of P such that $g' = f(g)$ and $ss'(f(j)) = f(ss(j))$ for all $j \in Tid$.*

Thus, with Theorems 5 and 6, we have successfully reduced the problem of k-bounded reachability on a dynamic concurrent pushdown system to a concurrent pushdown system with $k + 1$ threads.

6 Related work

We are not aware of any previous work that develops a theory of context-bounded analysis of concurrent software that is sound and complete up to the bound. Our techniques exploit results from model checking of sequential pushdown systems, in particular, Schwoon's generalization [27] of regular representation of sequential pushdown store languages [3, 12]. We have discussed the relation to our previous work on procedure summaries [20] and the KISS checker [22] in Section 1.

The notion of bounded-depth model checking, popular in hardware verification, can also be used for software verification [7]. These techniques bound the execution depth resulting in analysis of finite executions. In contrast, due to unbounded exploration within a thread context, our work allows analysis of unbounded execution sequences.

A number of model checkers have been developed for concurrent software [17, 14, 29, 9, 26, 18, 30]. All of these checkers keep explicit representation of the thread stacks, which might result in non-termination. Our analysis maintains a symbolic representation of the thread stacks and is guaranteed to terminate.

A variety of automated compositional techniques for verifying concurrent software have been developed [6, 16, 13, 15]. These techniques verify each process separately in an automatically constructed abstraction of the environment. The constructed abstraction is typically stackless and imprecise. As a result, these techniques are sound but not complete.

The idea of abstracting an unbounded number of processes into a single process has been used in verification of cache-coherence protocols [19] and compositional verification of software [15].

For restricted models of synchronization, assertion checking is decidable even with both concurrency and procedure calls. Esparza and Podelski present an algorithm for this restricted class of programs [11]. Alur and Grosu have studied the interaction between concurrency and procedure calls in the context of refinement between STATECHART programs [1]. At each step of the refinement process, their system allows either the use of nesting (the equivalent of procedures) or parallelism, but not both. Also, recursively nested modes are not allowed. In contrast, we place no restrictions on how parallelism interacts with procedure calls, and allow recursive procedures.

Bouajjani, Esparza, and Touili present an analysis that constructs abstractions of context-free languages [5]. The abstractions are chosen so that the emptiness of the intersection of the abstractions is decidable. Their analysis is sound but incomplete due to overapproximation in the abstractions.

7 Conclusion

In this paper we give for the first time a theory of context-bounded model checking for concurrent software that is sound up to the bound in the sense that it explores each context to full depth. Our algorithm finds any error that can possibly manifest itself in an error trace with a number of context switches within the bound, even in the presence of unbounded parallelism. It is an important research problem for future work to integrate our algorithm into explicit state model checking frameworks such as ZING [2].

References

1. R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. In *POPL 00: Principles of Programming Languages*, pages 390–402. ACM, 2000.
2. T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *CONCUR 2004: Fifteenth International Conference on Concurrency Theory, London, U.K., September 2004*, LNCS. Springer-Verlag, 2004. Invited paper.
3. J-M. Autebert, J. Berstel, and L. Boasson. Context-free languages and pushdown automata. In *Handbook of Formal Languages, vol. 1 (Eds.: G. Rozenberg and A. Salomaa)*, pages 111 – 174. Springer-Verlag, 1997.
4. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, January 2002.
5. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL 03: Principles of Programming Languages*, pages 62–73. ACM, 2003.
6. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.
7. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
8. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.
9. J. Corbett, M. Dwyer, John Hatcliff, Corina Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *ICSE 00: Software Engineering*, 2000.
10. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI: Programming Language Design and Implementation*, pages 57–69. ACM, 2002.

11. J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *POPL 00: Principles of Programming Languages*, pages 1–11. ACM, 2000.
12. A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9, 1997.
13. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE 02: Automated Software Engineering*, pages 3–12, 2002.
14. P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages*, pages 174–186, 1997.
15. T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI 04: Programming Language Design and Implementation*, pages 1–13, 2004.
16. T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV 03: Computer-Aided Verification*, pages 262–274, 2003.
17. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
18. M. Musuvathi, D. Park, A. Chou, D. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI 02: Operating Systems Design and Implementation*, 2002.
19. F. Pong and M. Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1):82–126, 1997.
20. S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL 04: ACM Principles of Programming Languages*, pages 245–255. ACM, 2004.
21. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. Technical Report MSR-TR-2004-70, Microsoft Research, 2004.
22. S. Qadeer and D. Wu. KISS: Keep it simple and seqeuential. In *PLDI 04: Programming Language Design and Implementation*, pages 14–24. ACM, 2004.
23. J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Fifth International Symposium on Programming*, Lecture Notes in Computer Science 137, pages 337–351. Springer-Verlag, 1981.
24. G. Ramalingam. Context sensitive synchronization sensitive analysis is undecidable. *ACM Trans. on Programming Languages and Systems*, 22:416–430, 2000.
25. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.
26. Robby, M. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *FSE 03: Foundations of Software Engineering*, pages 267–276. ACM, 2003.
27. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Lehrstuhl für Informatik VII der Technischen Universität München, 2000.
28. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.
29. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE 00: Automated Software Engineering*, pages 3–12, 2000.
30. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL 01: Principles of Programming Languages*, pages 27–40, 2001.