Context-Bounded Model Checking of Concurrent Software

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

July 2004

Technical Report
MSR-TR-2004-69

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

The design of concurrent programs is a complex endeavor. The main intellectual difficulty of this task lies in reasoning about the interaction between concurrently executing threads. Concurrency results in insidious programming errors that are difficult to reproduce and fix. Therefore, analysis techniques that can automatically detect and pinpoint errors in concurrent programs can be invaluable. In this paper, we present a novel interprocedural static analysis based on model checking [7, 23] for finding subtle safety errors in concurrent programs.

Concurrent programs are significantly more difficult to analyze than sequential programs. For example, the problem of checking assertions in a single-threaded boolean program with procedures (and consequently an unbounded stack) is decidable [29, 25]. In fact, the algorithm for solving this problem is the basis of a number of efficient static analysis tools [3, 9] for sequential programs. But the problem of checking assertions in a multithreaded boolean program with procedures is undecidable [24].

In the face of this undecidability, most analyses for concurrent programs presented so far have suffered from two limitations. Some analyses 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 5.

In this paper, we take a different approach to static verification of concurrent software 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. The reason for our belief is two fold. 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. In Section 1.1, we present an example taken from a Windows NT device driver to illustrate this point.

Our work is inspired by the KISS project and significantly extends its scope. 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. For example, KISS is unable to explore more than two context switches for a concurrent program with two threads. This paper presents a general algorithm for exploring an arbitrary number of context switches using an approach completely different from that of KISS.

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 [21] is not guaranteed to terminate for context-bounded model checking because it keeps an explicit representation of the stack of each thread. Summarization [21] 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 [2, 13, 28] 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 turn our attention to the problem of 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 tremendously increases the applicability of context-bounded model checking.

1.1 Example

In this section, we present an example to show that subtle bugs caused by concurrency can be manifested by executions with few contexts. Figure 1 shows code that is typically found in Windows NT device drivers for handling asynchronous IRP (Interrupt Request Packet) cancellation.

`DispatchRoutine` is a typical dispatch function of a device driver. The code for this function shows that a decision has been made to enqueue `irp` for future processing. First, the function `IoSetCancelRoutine` is called to set a cancellation routine for `irp`. The cancellation routine is called by the operating system if `irp` is cancelled. Then, `irp` is enqueued into a device queue under the protection of a spin lock. Finally, `irp` is marked pending and `STATUS_PENDING` is returned. While a thread is processing `irp` in `DispatchRoutine`, the operating system may cancel `irp` asynchronously by calling the function `IoCancelIrp` in another thread. This thread calls the cancel routine for `irp` if it is nonnull.

Consider the following execution. The first thread begins by calling `DispatchRoutine(obj,irp)`, sets the cancellation routine for `irp` to `CancelRoutine`, and proceeds until the call to `IoMarkIrpPending(irp)`. Just before this call, a

```
NTSTATUS
DispatchRoutine(
    DEVICE_OBJECT *obj,
    IRP *irp)
{
    DEVICE_EXTENSION *e =
           obj->DeviceExtension;
    .
    .
    .
    IoSetCancelRoutine(irp,
                       CancelRoutine);
    KeAcquireSpinLock(&e->SpinLock);
    EnqueueIrp(irp);
    KeReleaseSpinLock(&e->SpinLock);
    IoMarkIrpPending(irp);
    return STATUS_PENDING;
}

void CancelRoutine(
    DEVICE_OBJECT *obj,
    IRP *irp)
{
    DEVICE_EXTENSION *e =
           obj->DeviceExtension;
    IoReleaseCancelSpinLock();
    .
    .
    .
    KeAcquireSpinLock(&e->SpinLock);
    DequeueIrp(irp);
    KeReleaseSpinLock(&e->SpinLock);
    IoCompleteIrp(irp);
}

void IoCancelIrp(IRP *irp)
{
    DEVICE_OBJECT *obj;
    void (*Routine)(IRP *);
    IoAcquireCancelSpinLock();
    .
    .
    .
    Routine = IoSetCancelRoutine(irp,
                                 NULL);
    if (Routine != NULL) {
        Routine(irp);
    }
}
```

Figure 1: Example

context switch occurs and `IoCancelIrp(obj,irp)` begins execution in a second thread. This thread calls `CancelRoutine(obj,irp)`, which eventually calls `IoCompleteIrp(irp)`. A second context switch occurs and the first thread calls `IoMarkIrpPending(irp)`, thereby violating the following important rule governing the interface between the driver and the operating system.

> An IRP should not be marked pending after it has been completed.

Note that the erroneous execution had only three contexts (and two context switches). Additional experimental evidence for our belief that many errors manifest within a small number of contexts is provided by our previous work with the KISS static checker [22], where we found a variety of errors in systems code similar to the one shown above.

1.2 Overview

The remainder of this paper is organized as follows. In Section 2 we introduce sequential pushdown systems and the regularity of sequential pushdown store languages. In Section 3 we provide an algorithm for solving the context-bounded model checking problem for concurrent pushdown systems with an arbitrary fixed number of threads. We prove the algorithm correct and analyze its complexity. In Section 4 we consider dynamic concurrent pushdown systems with unbounded numbers of threads. We reduce the context-bounded model checking problem for such systems to context-bounded model checking for the systems studied in section 3 by introducing an abstraction. We prove the reduction sound and complete. Section 5 contains a discussion of related work, and Section 6 concludes. Appendix A contains proofs of theorems and lemmas whose proofs were left out of the main text.

2 Pushdown systems

Domains

γ	$\in$	Γ	*Stack alphabet*
w	$\in$	Γ^*	*Stack*
g	$\in$	G	*Global state*
Δ	$\subseteq$	$(G \times \Gamma) \times (G \times \Gamma^*)$	*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 [2, 13]. 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 [28]:

- 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 states reachable from S, i.e., $Post_\Delta^*(S) = \{c \mid \exists c' \in S.\ c' \longrightarrow_\Delta^* c\}$. The following theorem

shows that the set of reachable configurations from a regular set of configurations is again regular:

Theorem 1 ([28]) *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.*

The construction of A' from A is done [28] by adding states and transitions to A. Let g, g' range over G and let q range over Q. For every pair (g', γ') such that Δ contains a transition $(\langle g, \gamma\rangle, \langle g', \gamma'\gamma''\rangle)$ a new state $q_{g',\gamma'}$ is added. Three kinds of transitions are successively added. First, a transition (g', ϵ, q) is added whenever $(\langle g, \gamma\rangle, \langle g', \epsilon\rangle) \in \Delta$ and $q \in \delta^*(g, \gamma)$. Second, if $(\langle g, \gamma\rangle, \langle g', \gamma'\rangle) \in \Delta$ and $q \in \delta^*(g, \gamma)$ then a transition (g', γ', q) is added. Third, if $(\langle g, \gamma\rangle, \langle g', \gamma'\gamma'\rangle) \in \Delta$ and $q \in \delta^*(g, \gamma)$ then transitions $(g', \gamma', q_{g',\gamma'})$ and $(q_{g',\gamma'}, \gamma'', q)$ are added. For more details on this construction we refer the reader to [28].

3 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., $\longrightarrow_P = \bigcup_{i=0}^{N} \longrightarrow_i$.

3.1 Example

To illustrate how a concurrent pushdown system can be used to represent programs, we present a simple program with two threads in Figure 2. We represent this program as a concurrent pushdown system $P = (G, \Gamma, \Delta_0, \Delta_1, g_{in}, w_{in})$. We let G be the set of valuations to global variables `m` and `task`. There is a special local variable `pc`, the program counter, which takes values from the set of program labels $PC = \{\texttt{L0}, \texttt{L1}, \texttt{L2}, \ldots, \texttt{L8}\}$. `L0` is a special label not present in the program used to initialize the stack of each thread. We let Γ be the set of valuations to `pc`. We write $(m, task)$ to denote the valuation assigning the value m to `m` and the value *task* to `task`. Hence, a transition $(\langle g, \gamma\rangle, \langle g', w\rangle)$ has the form

$$(\langle (m, task), pc\rangle \ , \ \langle (m', task'), w\rangle)$$

where $w \in PC^*$.

```
boolean m = 0, task = 0;

Thread 0                Thread 1
while(1) {              while(1) {
L1:  wait(¬m);          L4:  wait(m);
L2:  InitTask();        L5:  DoTask();
L3:  m := 1;            L6:  m := 0;
}                       }

InitTask() {            DoTask() {
L7:  task := {0,1};     L8:  task := ¬task;
}                       }
```

Figure 2: Concurrent pushdown system.

The initial global state $g_{in} = (\texttt{0}, \texttt{0})$. The initial stack configuration $w_{in} = \texttt{L0}$. We introduce two initial transitions $(I0)$ and $(I1)$ for Δ_0 and Δ_1 respectively.

$$\begin{array}{ll} (I0) & (\langle (0,0), \texttt{L0}\rangle) \ , \ (\langle (0,0), \texttt{L1}\rangle) \\ (I1) & (\langle (0,0), \texttt{L0}\rangle \ , \ \langle (0,0), \texttt{L4}\rangle) \end{array}$$

Thread 0 and thread 1 use transitions $(I0)$ and $(I1)$ to move to their first instructions respectively.

The remainder of the transitions in Δ_0 are shown below.

$$\begin{array}{ll} (1) & (\langle (\texttt{0}, task), \texttt{L1}\rangle \ , \ \langle (\texttt{0}, task), \texttt{L2}\rangle) \\ (2) & (\langle (m, task), \texttt{L2}\rangle \ , \ \langle (m, task), \texttt{L7} \cdot \texttt{L3}\rangle) \\ (3) & (\langle (m, task), \texttt{L3}\rangle \ , \ \langle (\texttt{1}, task), \texttt{L1}\rangle) \\ (4) & (\langle (m, task), \texttt{L7}\rangle \ , \ \langle (m, \texttt{0}), \epsilon\rangle) \\ (5) & (\langle (m, task), \texttt{L7}\rangle \ , \ \langle (m, \texttt{1}), \epsilon\rangle) \end{array}$$

We have used a compact notation in describing the transitions above. Each of (1) through (5) denotes a set of transitions for each value of m and *task*, where m and *task* range over the set $\{0, 1\}$.

The transition relation Δ_1 for thread 1 can be defined analogously. Further examples of similar program representations can be found in [28].

3.2 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:

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

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 $c_0 \xrightarrow{k} c_1$?

3.3 Aggregate configurations

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

$$\langle\!\langle g, R \rangle\!\rangle = \{\langle g, w \rangle \mid w \in R\}$$

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:

$$S = \biguplus_{i=1}^{m} \langle\!\langle g_i, R_i \rangle\!\rangle \tag{1}$$

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 (1), 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 (1), 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 $j \in \{1, \ldots, m\}$.

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:

$$\begin{array}{l} \langle\!\langle g, R_0, \ldots, R_i, \ldots, R_N \rangle\!\rangle \Longrightarrow_i \\ \langle\!\langle g', R_0, \ldots, R_i', \ldots, R_N \rangle\!\rangle \end{array}$$

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 a concurrent pushdown system $P = (G, \Gamma, \Delta_0, \ldots, \Delta_N, g_{in}, w_{in})$ be given. Then, for any k,*

$$\langle g, w_0, \ldots, w_N \rangle \xrightarrow{k} \langle g', w_0', \ldots, w_N' \rangle$$

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 $i \in \{0, \ldots, N\}$.

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

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

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

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

Output : *Reach*

Figure 3: Algorithm

3.4 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 3.

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 6 of the algorithm in Figure 3, 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 3 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 [28] (described after Theorem 1) uses elements og 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. Because of this renaming, successive pushdown store automata constructed by the algorithm in Figure 3 may grow for increasing values of the bound k. The fact that the pushdown store automata could grow unboundedly for ever increasing k underlies the undecidability of the unbounded reachability problem.

Theorem 3 *Let a concurrent pushdown system* $P = (G, \Gamma, \Delta_0, \ldots, \Delta_N, g_{in}, w_{in})$ *be given. For any* k*, the algorithm in Figure 3 terminates on input* P *and* k*, and*

$$\langle\!\langle g_{in}, \{w_{in}\}, \ldots, \{w_{in}\}\rangle\!\rangle \stackrel{k}{\Longrightarrow} \langle\!\langle g', R'_0, \ldots, R'_N\rangle\!\rangle$$

if and only if the algorithm outputs Reach with $\langle g', A'_0, \ldots, A'_N\rangle \in Reach$ *such that and* $L(A'_i) = \langle\!\langle g', R'_i\rangle\!\rangle$ *for all* $i \in \{0, \ldots, N\}$.

PROOF Termination of the algorithm follows from the fact that the ADD statement in line 9 can only be executed a bounded number of times. Let $Count(i)$ denote the number of times ADD is called in line 9 with index argument i. Then clearly, $Count(0) = 1$ and $Count(i+1) = N \times |G| \times Count(i)$, since

the loops in line 6 and 8 are bounded by N and $|G|$ respectively. Hence, $Count(i+1) \leq (N \times |G|)^i$ is bounded for all i. Since an item is removed in each iteration of the **while** loop at line 4, it follows that the worklist eventually becomes empty and the algorithm terminates.

The algorithm computes reachability for the given pushdown system with k-bounded contexts by computing the reachability relation $\stackrel{k}{\Longrightarrow}$ over aggregate configurations represented by pushdown store automata. Reachable aggregate configurations $\langle g', R'_0, \ldots, R'_N \rangle$ are represented by pushdown store automata in the worklist items by the vectors $\langle g', A'_0, \ldots, A'_N \rangle$ and are collected in the set *Reach*. The theorem follows easily from Theorem 1, which implies that the automata constructed in line 7 of the algorithm correctly represent components of aggregate configurations. □

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

For a consurrent pushdown system $P = (G, \Gamma, \Delta_0, \ldots, \Delta_N, g_{in}, w_{in})$ we measure the size of P by $|P| = \max(|G|, |\Delta|, |\Gamma|)$. 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 3 decides the k-bounded reachability problem in time $\mathcal{O}(k^3(N|G|)^k|P|^5)$.*

PROOF From Theorem 3.5 of [28] it follows that the $Post^*$ operation on a set represented by A can be computed in time $\mathcal{O}(|P|^2|A| + |P|^3)$. Let $A(i)$ denote the size of automaton A as extended by successive applications of the $Post^*$ operator thorughout the iterations of the algorithm. It can be seen that $A(i) \leq c + i|P| \times i|P| \times |P| = \mathcal{O}(i^2|P|^3)$, where the terms $i|P|$ come from the number of states of the automaton, and $|P|$ bounds the number of labeled edges in its transition relation. Therefore, the time to compute $Post^*$ at index i is bounded by $Cost(i) = \mathcal{O}(|P|^2 \times i^2|P|^3 + |P|^3) = \mathcal{O}(i^2|P|^5)$. With $Items(i)$ denoting the number of work items with index i created throughout a run of the algorithm, we have $Items(0) = 1$ and $Items(i+1) = N|G| \times Items(i) = (N|G|)^i$. Summing over $i = 1 \ldots k$ iterations, the total cost of the algorithm is dominated by $\sum_{i=1}^{k}(N|G|)^i \times Cost(i) = \mathcal{O}(k^3(N|G|)^k|P|^5)$. □

4 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

ss	$\in$	$Stacks = Tid \to (\Gamma \cup \{\$\})^*$	
c	$\in$	$C = G \times Tid \times Stacks$	*Configuration*
$\rightsquigarrow$	$\subseteq$	$C \times C$	

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

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

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

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

All rules are guarded by the condition $t \le n$ 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.

```
boolean task = 0;

Thread 0
Tid t;
while(1) {
L1:  InitTask();
L2:  t := async DoTask();
L3:  join(t);
}

InitTask() {
L7:  task := {0,1};
}

DoTask() {
L8:  task := ¬task;
}
```

Figure 4: Dynamic concurrent pushdown system.

4.1 Example

Figure 4 shows an example of dynamic thread creation. This example replaces the mutex-based synchronization of the example in Figure 2 with fork-join synchronization. To model dynamic thread creation, we introduce a (ghost) global variable `inpc` which takes values from the set PC. This variable contains the program label of a thread when it is forked. We represent this program as a dynamic concurrent pushdown system

$$(\{\texttt{task}, \texttt{inpc}\}, \emptyset, \{\texttt{pc}\}, \{\texttt{t}\}, \Delta, \Delta_F, \Delta_J, g_{in}, \gamma_{in}).$$

The initial global state $g_{in} = (\texttt{0}, \texttt{L1})$. The initial stack configuration $w_{in} = (0, \texttt{L0})$. We introduce an initial transition (I) in Δ.

$$(I) \quad (\langle (task, inpc), (0, \texttt{L0})\rangle \ , \ \langle (task, \texttt{L0}), (0, inpc)\rangle)$$

Each thread uses transitions (I) to move to its first instruction. The remainder of the transitions in Δ are shown below.

(1) $(\langle (task, inpc), (t, \texttt{L1})\rangle \ , \ \langle (task, inpc), (0, \texttt{L7}) \cdot (t, \texttt{L2})\rangle)$
(2) $(\langle (task, inpc), (t, \texttt{L7})\rangle \ , \ \langle (0, inpc), \epsilon\rangle)$
(3) $(\langle (task, inpc), (t, \texttt{L7})\rangle \ , \ \langle (1, inpc), \epsilon\rangle)$
(4) $(\langle (task, inpc), (t, \texttt{L8})\rangle \ , \ \langle (\neg task, inpc), \epsilon\rangle)$

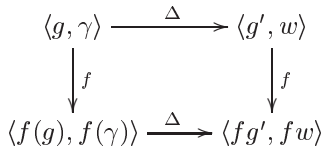

Figure 5: Pictorial view of assumption A1.

We have used a compact notation similar to the example in Section 3.1 in describing the transitions above.

The transitions in Δ_F are shown below.

$$(1)\quad (\langle tid, (task, \texttt{L0}), (t, \texttt{L2})\rangle\ , \\ \langle (task, \texttt{L8}), (tid, \texttt{L3})\rangle)$$

The transitions in Δ_J are shown below.

$$(1)\quad (\langle \texttt{t}, (task, inpc), (t, \texttt{L3})\rangle\ , \\ \langle (task, inpc), (t, \texttt{L1})\rangle)$$

4.2 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.

First, we formalize the assumptions about the restricted use of thread identifiers. 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. We now formalize our assumptions about the transition relations Δ, Δ_F, and Δ_J.

A1. For all $g \in G$, $\gamma \in \Gamma$, and renaming functions f such that $f(g)$ and $f(\gamma)$ are defined, the following statements are true.

1. If $(\langle g, \gamma\rangle, \langle g', w\rangle) \in \Delta$ then there exist $fg' \in G$ and $fw \in \Gamma^*$ such that $fg' = f(g')$, $fw = f(w)$, and $(\langle f(g), f(\gamma)\rangle, \langle fg', fw\rangle) \in \Delta$.
2. If $(\langle f(g), f(\gamma)\rangle, \langle fg', fw\rangle) \in \Delta$ then there exist $g' \in G$ and $w \in \Gamma^*$ such that $fg' = f(g')$, $fw = f(w)$, and $(\langle g, \gamma\rangle, \langle g', w\rangle) \in \Delta$.

Figure 5 presents a pictorial view of assumpton A1. This view shows four arrows, two horizontal labeled with the transition relation Δ and two vertical labeled with the renaming function f. The assumption expresses a requirement on tuples (g, γ) for which the left vertical arrow holds. A1.1 states that if the top horizontal arrow holds in addition, then the remaining two arrows hold. Conversely, A1.2 states that if the bottom horizontal arrow holds in addition, then the remaining two arrows hold. A careful reader can verify that the transition relation Δ in Section 4.1 satisfies A1 by verifying it on each transition (I) and $(1) - (4)$ separately.

A2. For all $t \in Tid$, $g \in G$, $\gamma \in \Gamma$, and renaming functions f such that $f(t)$, $f(g)$, and $f(\gamma)$ are defined, the following statements are true.

1. If $(t, \langle g, \gamma\rangle, \langle g', w\rangle) \in \Delta_F$ then there exist $fg' \in G$ and $fw \in \Gamma^*$ such that $fg' = f(g')$, $fw = f(w)$, and $(f(t), \langle f(g), f(\gamma)\rangle, \langle fg', fw\rangle) \in \Delta_F$.
2. If $(f(t), \langle f(g), f(\gamma)\rangle, \langle fg', fw\rangle) \in \Delta_F$ then there exist $g' \in G$ and $w \in \Gamma^*$ such that $fg' = f(g')$, $fw = f(w)$, and $(t, \langle g, \gamma\rangle, \langle g', w\rangle) \in \Delta_F$.

A3. For all $x \in LTV$, $g \in G$, $\gamma \in \Gamma$, and renaming functions f such that $f(g)$ and $f(\gamma)$ are defined, the following statements are true.

1. If $(x, \langle g, \gamma\rangle, \langle g', w\rangle) \in \Delta_J$ then there exist $fg' \in G$ and $fw \in \Gamma^*$ such that $fg' = f(g')$, $fw = f(w)$, and $(x, \langle f(g), f(\gamma)\rangle, \langle fg', fw\rangle) \in \Delta_J$.
2. If $(x, \langle f(g), f(\gamma)\rangle, \langle fg', fw\rangle) \in \Delta_J$ then there exist $g' \in G$ and $w \in \Gamma^*$ such that $fg' = f(g')$, $fw = f(w)$, and $(x, \langle g, \gamma\rangle, \langle g', w\rangle) \in \Delta_J$.

Assumption A2 about Δ_F and A3 about Δ_J are similar in spirit to Δ and could be verified on the example in Section 4.1

4.3 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 3.

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

$$\begin{array}{rcl} P_k & = & ((AbsG_k \times Tid_k \times \mathcal{P}(Tid_k)), \\ & & (Abs\Gamma_k \cup \{\$\}), \\ & & \Delta_0, \Delta_1, \ldots, \Delta_k, \\ & & (g_{in}, 0, \emptyset), \\ & & \gamma_{in}\$). \end{array}$$

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. Finally, 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 $f : Tid \to Tid_k$ 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 $f : Tid \to Tid_k$ 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.

5 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 [28] of regular representation of sequential pushdown store languages [2, 13]. We have discussed the relation to our previous work on procedure summaries [21] and the KISS checker [22] in Section 1. We discuss other related work under the following headings.

Bounded-depth model checking. The notion of bounded-depth model checking, popular in hardware verification, can also be used for software verification [6]. 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.

Software model checking. A number of model checkers have been developed for concurrent software [18, 15, 30, 8, 27, 19, 31]. 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.

Flow-based analyses. Duesterwald and Soffa use a system of dataflow equations to check if two statements in a concurrent program can potentially execute in parallel [10]. Their analysis is conservative and restricted to Ada rendezvous constructs. Dwyer and Clarke check properties of concurrent programs by dataflow analysis, but use inlining to flatten procedure calls [11]. Flow-insenstive analyses are independent of the ordering between program statements and can be generalized easily to multithreaded programs with procedure calls. Rinard presents a survey of techniques for analysis of concurrent programs [26].

Compositional reasoning. A variety of automated compositional techniques for verifying concurrent software have been developed [5, 17, 14, 16]. 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.

Counting abstraction. The idea of abstracting an unbounded number of processes into a single process has been used in verification of cache-coherence protocols [20] and compositional verification of software [16].

Restricted synchronization models. 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 [12]. 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.

Other work. Bouajjani, Esparza, and Touili present an analysis that constructs abstractions of context-free languages [4]. 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.

6 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.

The context-bounded model checking problem does not appear to be solvable by simply introducing a counter, and our techniques are very different from other bounding approaches, such as depth-bounded model checking or the KISS system. The main challenge for devising an algorithm that is sound up to the bound is that an unbounded number of configurations must be stored for each context in order to allow search to continue from any reachable configuration within the current context after a context switch. We solve the problem by exploiting the regularity of pushdown store languages [2, 13] and the fact that the configurations reachable in a single pushdown system from a regular set of configurations is again regular [28]. This result allows us to finitely represent repeated applications of the $Post^*$ operator on a concurrent pushdown system by vectors of regular automata. We extend the technique to dynamic concurrent pushdown systems by abstracting an unbounded number of threads, thereby reducing the context-bounded reachability problem for dynamic systems to the non-dynamic case.

We believe that the theory provided by this paper holds the promise that we may devise practical model checking for complex concurrent systems. Experience suggests that many concurrency errors manifest themselves in some execution with few contexts. Perhaps it can even be held that most well-designed systems ought to be amenable to exhaustive analysis with a bounded number of contexts, since properties of a system that depend on unbounded numbers of contexts could well be beyond human comprehension.

It is an important research problem for future work to integrate our algorithm into explicit state model checking frameworks. Many, if not most, existing software model checkers are explicit state, and the challenge is that our algorithm uses a symbolic representation of aggregate stack configurations.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, pages 57–69. ACM, 2002.

[10] E. Duesterwald and M. L. Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *TAV 91: Testing, Analysis and Verification*, pages 36–48. ACM, 1991.

[11] M. Dwyer and L. Clarke. Data flow analysis for verifying properties of concurrent programs. In *FSE 94: Foundations of Software Engineering*, pages 62–75. ACM, 1994.

[12] 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.

[13] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9, 1997.

[14] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE 02: Automated Software Engineering*.

[15] P. Godefroid. Model checking for programming languages using verisoft. In *POPL 97: Principles of Programming Languages*, pages 174–186, 1997.

[16] 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.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV 03: Computer-Aided Verification*, pages 262–274, 2003.

[18] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[19] 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.

[20] F. Pong and M. Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1):82–126, 1997.

[21] 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.

[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] M. C. Rinard. Analysis of multithreaded programs. In *SAS 01: Static Analysis*, LNCS 2126, pages 1–19. Springer-Verlag, 2001.

[27] 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.

[28] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Lehrstuhl für Informatik VII der Technischen Universität München, 2000.

[29] 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.

[30] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE 00: Automated Software Engineering*, pages 3–12, 2000.

[31] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL 01: Principles of Programming Languages*, pages 27–40, 2001.

A Proofs

A.1 Proof of Theorem 2

Lemma 1 *Suppose that*

$$\langle g, w_0, \ldots, w_i, \ldots, w_N \rangle \longrightarrow_i^* \langle g', w_0, \ldots, w_i', \ldots, w_N \rangle$$

with $\langle g, w_i \rangle \in \langle\!\langle g, R_i \rangle\!\rangle$. *Then there exists* R' *such that* $\langle\!\langle g', R_i' \rangle\!\rangle \in Post^*_{\Delta_i}(\langle\!\langle g, R_i \rangle\!\rangle)$ *and* $\langle g', w_i' \rangle \in \langle\!\langle g', R_i' \rangle\!\rangle$.

PROOF The assumption implies that

$$\langle g, w_i\rangle \longrightarrow^*_{\Delta_i} \langle g', w_i'\rangle$$

hence

$$\langle g', w_i'\rangle \in Post^*_{\Delta_i}(\langle\!\langle g, R_i\rangle\!\rangle) \tag{2}$$

Writing $Post^*_{\Delta_i}(\langle\!\langle g, R_i\rangle\!\rangle)$ on the form (1), we have

$$Post^*_{\Delta_i}(\langle\!\langle g, R_i\rangle\!\rangle) = \biguplus_{j=1}^{m} \langle\!\langle g_j, R_j'\rangle\!\rangle$$

for some $R_1', \ldots, R_m'$, where $G = \{g_1, \ldots, g_m\}$. Since $g' = g_j$ for some $j \in \{1, \ldots, m\}$, it follows from (2) that $\langle g', w_i'\rangle \in \langle\!\langle g_j, R_j'\rangle\!\rangle$, and we can choose $R' = R_j'$. □

Lemma 2 *Suppose that*

$$\langle g, w_0, \ldots, w_N\rangle \xrightarrow{k} \langle g', w_0, \ldots, w_N\rangle$$

Then

$$\langle\!\langle g, \{w_0\}, \ldots, \{w_N\}\rangle\!\rangle \stackrel{k}{\Longrightarrow} \langle\!\langle g', R_0', \ldots, R_N'\rangle\!\rangle$$

for some $R_0', \ldots, R_N'$ such that $w_i' \in R_i'$ for all $i \in \{0, \ldots, N\}$.

PROOF By induction on $k \geq 1$. For the base case, $k = 1$, the assumption is that

$$\langle g, w_0, \ldots, w_N\rangle \longrightarrow^*_i \langle g', w_0, \ldots, w_N\rangle$$

for some $i \in \{0, \ldots, N\}$, i.e.,

$$\langle g, w_0, \ldots, w_i, \ldots, w_N\rangle \longrightarrow^*_i \langle g', w_0, \ldots, w_i', \ldots, w_N\rangle$$

Since $\langle g, w_i\rangle \in \langle\!\langle g, \{w_i\}\rangle\!\rangle$, Lemma 1 implies that $\langle g', w_i'\rangle \in \langle\!\langle g', R'\rangle\!\rangle$ for some R' such that $\langle\!\langle g', R'\rangle\!\rangle \in Post^*_{\Delta_i}(\langle\!\langle g, \{w_i\}\rangle\!\rangle)$. By the definition of $\Longrightarrow_i$, this shows that

$$\begin{array}{l}\langle\!\langle g, \{w_0\}, \ldots, \{w_i\}, \ldots, \{w_N\}\rangle\!\rangle \Longrightarrow_i \\ \langle\!\langle g, \{w_0\}, \ldots, R', \ldots, \{w_N\}\rangle\!\rangle\end{array}$$

which proves the lemma in the base case.

For the inductive case, assume

$$\begin{array}{l}\langle g, w_0, \ldots, w_i, \ldots, w_N\rangle \longrightarrow^{[k+1]} \\ \langle g'', w_0'', \ldots, w_i'', \ldots, w_N''\rangle\end{array}$$

by the relations

$$\begin{array}{l}\langle g, w_0, \ldots, w_i, \ldots, w_N\rangle \longrightarrow^{[k]} \\ \langle g', w_0', \ldots, w_i', \ldots, w_N'\rangle\end{array} \tag{3}$$

and

$$\begin{array}{l}\langle g', w_0', \ldots, w_i', \ldots, w_N'\rangle \longrightarrow^*_i \\ \langle g'', w_0', \ldots, w_i'', \ldots, w_N'\rangle\end{array} \tag{4}$$

By induction hypothesis and (3), we have

$$\begin{array}{l}\langle\!\langle g, \{w_0\}, \ldots, \{w_i\}, \ldots, \{w_N\}\rangle\!\rangle \stackrel{k}{\Longrightarrow} \\ \langle\!\langle g', R_0', \ldots, R_i', \ldots, R_N'\rangle\!\rangle\end{array} \tag{5}$$

for some $R_0', \ldots, R_N'$ such that $w_j' \in R_j'$ for all $j \in \{0, \ldots, N\}$. Since we have $\langle g', w_i'\rangle \longrightarrow^*_{\Delta_i} \langle g'', w_i''\rangle$ by (4) and $w_i' \in R_i'$, it follows that $\langle\!\langle g', w_i'\rangle\!\rangle \in \langle\!\langle g', R_i'\rangle\!\rangle$. Lemma 1 then shows that we have $\langle g'', w_i''\rangle \in \langle\!\langle g'', R_i''\rangle\!\rangle$ for some R_i'' such that $\langle\!\langle g'', R_i''\rangle\!\rangle \in Post^*_{\Delta_i}(\langle\!\langle g', R_i'\rangle\!\rangle)$. It follows that we have

$$\begin{array}{l}\langle\!\langle g', R_0', \ldots, R_i', \ldots, R_N'\rangle\!\rangle \Longrightarrow_i \\ \langle\!\langle g'', R_0', \ldots, R_i'', \ldots, R_N'\rangle\!\rangle\end{array} \tag{6}$$

Composing (5) and (6) yields

$$\begin{array}{l}\langle\!\langle g, \{w_0\}, \ldots, \{w_i\}, \ldots, \{w_N\}\rangle\!\rangle \stackrel{k}{\Longrightarrow} \\ \langle\!\langle g'', R_0', \ldots, R_i'', \ldots, R_N'\rangle\!\rangle\end{array}$$

thereby proving the lemma in the inductive case. □

Lemma 3 *Suppose that*

$$\langle\!\langle g, R_0, \ldots, R_i, \ldots, R_N\rangle\!\rangle \Longrightarrow_i \langle\!\langle g', R_0, \ldots, R_i', \ldots, R_N\rangle\!\rangle$$

Then, for all $w_i' \in R_i'$, there exists $w_i \in R_i$ such that

$$\langle g, w_0, \ldots, w_i, \ldots, w_N\rangle \longrightarrow^*_i \langle g', w_0, \ldots, w_i', \ldots, w_N\rangle$$

for all $w_j \in R_j$, $j \in \{0, \ldots, i-1, i+1, \ldots, N\}$.

PROOF By the assumption we have

$$\langle\!\langle g', R_i'\rangle\!\rangle \in Post^*_{\Delta_i}(\langle\!\langle g, R_i\rangle\!\rangle)$$

Hence, we can write

$$\begin{array}{l}\langle\!\langle g', R_i'\rangle\!\rangle = \\ \{\langle g', w_i'\rangle \mid \exists \langle g, w_i\rangle \in \langle\!\langle g, R_i\rangle\!\rangle.\ \langle g, w_i\rangle \longrightarrow^*_{\Delta_i} \langle g', w_i'\rangle\}\end{array}$$

which shows that, for all $w_i' \in R_i'$, there exists w_i such that

$$\langle g, w_i\rangle \longrightarrow^*_{\Delta_i} \langle g', w_i'\rangle$$

The lemma now follows from the definition of the transition relation $\longrightarrow^*_i$. □

Lemma 4 *Let strings $w_0, \ldots, w_N$ and $w'_0, \ldots, w'_N$ be given. Assume that*

$$\langle\!\langle g, \{w_0\}, \ldots, \{w_N\} \rangle\!\rangle \stackrel{k}{\Longrightarrow} \langle\!\langle g', R'_0, \ldots, R'_N \rangle\!\rangle$$

for some $R'_0, \ldots, R'_N$ such that $w'_i \in R'_i$ for all $i \in \{0, \ldots, N\}$. Then

$$\langle g, w_0, \ldots, w_N \rangle \stackrel{k}{\longrightarrow} \langle g', w'_0, \ldots, w'_N \rangle$$

PROOF By induction on $k \geq 1$. For the base case, $k = 1$, the assumption yields:

$$\begin{array}{l} \langle\!\langle g, \{w_0\}, \ldots, \{w_i\}, \ldots, \{w_N\} \rangle\!\rangle \Longrightarrow_i \\ \langle\!\langle g', \{w_0\}, \ldots, R'_i, \ldots, \{w_N\} \rangle\!\rangle \end{array}$$

for some i and R'_i with $w'_i \in R'_i$. Then, by Lemma 3, there exists $w''_i \in \{w_i\}$ such that

$$\langle g, w_1, \ldots, w''_i, \ldots, w_N \rangle \longrightarrow_i^* \langle g, w_1, \ldots, w'_i, \ldots, w_N \rangle$$

holds. Since we must have $w''_i = w_i$ by the choice of w''_i, the lemma follows.

For the inductive case, assume

$$\langle\!\langle g, \{w_0\}, \ldots, \{w_N\} \rangle\!\rangle \stackrel{k+1}{\Longrightarrow} \langle\!\langle g', R'_0, \ldots, R'_N \rangle\!\rangle$$

with $w'_i \in R'_i$ for all $i \in \{0, \ldots, N\}$. Then, for some $j \in \{0, \ldots, N\}$ and g'', we have

$$\begin{array}{l} \langle\!\langle g, \{w_0\}, \ldots, \{w_N\} \rangle\!\rangle \stackrel{k}{\Longrightarrow} \\ \langle\!\langle g'', R'_0, \ldots, R'_j, \ldots, R'_N \rangle\!\rangle \end{array} \tag{7}$$

and, for some $j \in \{0, \ldots, N\}$ and R'', we have

$$\begin{array}{l} \langle\!\langle g'', R'_0, \ldots, R'_{j-1}, R'', R'_{j+1} \ldots, R'_N \rangle\!\rangle \Longrightarrow_j \\ \langle\!\langle g', R'_0, \ldots, R'_{j-1}, R'_j, R'_{j+1}, \ldots, R'_N \rangle\!\rangle \end{array} \tag{8}$$

By induction hypothesis and (7) we have

$$\begin{array}{l} \langle g, w_0, \ldots, w_n \rangle \stackrel{k}{\longrightarrow} \\ \langle g'', w'_0, \ldots, w'_{j-1}, w'', w'_{j+1}, \ldots, w'_N \rangle \end{array} \tag{9}$$

for all $w'' \in R''$. Lemma 3 applied to (8) implies that, for all $w'_j \in R'_j$ there exists $w''' \in R''$ such that

$$\begin{array}{l} \langle g'', w'_0, \ldots, w'_{j-1}, w''', w'_{j+1}, \ldots, w'_N \rangle \longrightarrow_j^* \\ \langle g', w'_0, \ldots, w'_{j-1}, w'_j, w'_{j+1}, \ldots, w'_N \rangle \end{array} \tag{10}$$

holds for all $w'_i \in R'_i$ for $i \in \{0, \ldots, j-1, j+1, \ldots, N\}$. Because (9) holds for all $w'' \in R''$, it holds for w''' in particular. Hence, composing (9) and (10) proves the lemma. □

Theorem 2 *Let a concurrent pushdown system $P = (G, \Gamma, \Delta_0, \ldots, \Delta_N, \langle g_{in}, w_{in} \rangle)$ be given. Then, for any k,*

$$\langle g, w_0, \ldots, w_N \rangle \stackrel{k}{\longrightarrow} \langle g', w'_0, \ldots, w'_0 \rangle$$

if and only if

$$\langle\!\langle g, \{w_0\}, \ldots, \{w_N\} \rangle\!\rangle \stackrel{k}{\Longrightarrow} \langle\!\langle g', R'_0, \ldots, R'_N \rangle\!\rangle$$

for some $R'_0, \ldots, R'_N$ such that $w'_i \in R'_i$ for all $i \in \{0, \ldots, N\}$.

PROOF The theorem follows directly from Lemma 2 and Lemma 4. □

A.2 Proof of Theorems 5 and 6

We fix a dynamic concurrent pushdown system P, a positive integer k, and the concurrent pushdown system P_k. In order to make the proofs easy, we augment each configuration $\langle g, n, ss \rangle$ of P to $\langle c, g, n, ss \rangle$, where c is a subset of *Tid*. This subset predicts at the beginning of an execution the identifiers of the threads that may perform a transition at any point in the execution. It is initialized nondeterministically and remains unchanged throughout the execution of the system. We extend the transition relation $\rightsquigarrow_t$ in the obvious way so that if $\langle c, g, n, ss \rangle \rightsquigarrow_t \langle c, g', n', ss' \rangle$ then $t \in c$. Clearly, every execution over the original configurations can be matched by an execution over the augmented configurations by choosing an appropriate initial value of c. In addition, for every k-bounded execution, a choice of c can be made such that $0 \in c$ and $|c| \leq k$. Henceforth, we assume that the set c in the augmented configurations of k-bounded execution sequences satisfies the above property.

Lemma 5 *For every reachable configuration $\langle c, g, n, ss \rangle$ of P, we have $ss(t) = \gamma_0\$$ for all $t \in Tid$ such that $t > n$.*

PROOF The proof is by induction on the length of the execution sequence leading to $\langle c, g, n, ss \rangle$. □

Lemma 6 *For every reachable configuration $\langle (g, n, \alpha), ss \rangle$ of P_k, we have $ss(t) = \gamma_0\$$ for all $t \in Tid_k$ such that $t > n$.*

PROOF The proof is by induction on the length of the execution sequence leading to $\langle (g, n, \alpha), ss \rangle$. □

Lemma 7 *For every reachable configuration $\langle (g, n, \alpha), ss\rangle$ of P_k, we have $t \in \alpha$ iff $ss(t) = \epsilon$ for all $t \in Tid_k$.*

PROOF The proof is by induction on the length of the execution sequence leading to $\langle (g, n, \alpha), ss\rangle$. □

Lemma 8 *For every k-bounded execution $\langle c, g_0, n_0, ss_0\rangle \rightsquigarrow_{t_1} \langle c, g_1, n_1, ss_1\rangle \rightsquigarrow_{t_2} \ldots \rightsquigarrow_{t_l} \langle c, g_l, n_l, ss_l\rangle$ of P, there is a renaming function $f : Tid \to Tid_k$ and an execution $\langle (g'_0, n'_0, \alpha_0), ss'_0\rangle \longrightarrow_{t'_1} \langle (g'_1, n'_1, \alpha_1), ss'_1\rangle \longrightarrow_{t'_2} \ldots \longrightarrow_{t'_l} \langle (g'_l, n'_l, \alpha_l), ss'_l\rangle$ of P_k such that the following statements are true.*

1. *$dom(f) = \{t \in Tid \mid 0 \leq t \leq n_l\}$.*
2. *The function f maps $dom(f) \cap c$ one-one onto $\{t \in Tid \mid 0 \leq t \leq n'_l\}$ and $dom(f) \setminus c$ to k.*
3. *$g'_l = f(g_l)$.*
4. *For all $t \in dom(f)$, $ss'_l(f(t)) = f(ss_l(t))$.*

PROOF We do the proof by induction over the length l of the k-bounded execution sequence of the dynamic concurrent pushdown system.

Base case ($l = 0$): Let f be such that $f(0) = 0$ and f is undefined elsewhere. We know that $n_0 = 0$ and $n'_0 = 0$. Therefore $dom(f) = \{t \in Tid \mid 0 \leq t \leq n_l\}$ and f maps $dom(f)$ one-one onto $\{t \in Tid \mid 0 \leq t \leq n'_l\}$. From the property of g_0, we get that $g'_0 = f(g_0) = g_0$. From the property of γ_0, we get that $\gamma'_0 = f(\gamma_0) = \gamma_0$. Therefore $ss'_0(f(0)) = ss'_0(0) = \gamma_0\$ = ss_0(0)$.

Inductive case: We assume that the theorem holds for l with a renaming function f. We now prove the theorem for $l + 1$ by a case analysis.

(SEQ): We have $\langle c, g_l, n_l, ss_l\rangle \rightsquigarrow_{t_{l+1}} \langle c, g_{l+1}, n_l, ss_l[t_{l+1} := ww']\rangle$, where $t_{l+1} \in c$, $t_{l+1} \leq n_l$, $ss_l(t_{l+1}) = \gamma w'$, and $(\langle g_l, \gamma\rangle, \langle g_{l+1}, w\rangle) \in \Delta$. By the induction hypothesis, we have $dom(f) = \{t \in Tid \mid 0 \leq t \leq n_l\}$. Therefore $t_{l+1} \in dom(f) \cap c$ and by the induction hypothesis, we get $f(t_{l+1}) \leq n'_l$. Let $t'_{l+1} = f(t_{l+1})$. By Assumption A1.1, we have $(\langle f(g_l), f(\gamma)\rangle, \langle f(g_{l+1}), f(w)\rangle) \in \Delta$. By rule (ABSSEQ), we get $(\langle (f(g_l), n'_l, \alpha_l), f(\gamma)\rangle\ ,\ \langle (f(g_{l+1}), n'_l, \alpha_l), f(w)\rangle) \in \Delta_{t'_{l+1}}$. By the induction hypothesis, we have $ss'_l(t'_{l+1}) = f(ss_l(t_{l+1})) = f(\gamma)f(w')$. Therefore, we have $\langle (f(g_l), n'_l, \alpha_l), ss'_l\rangle \longrightarrow_{t'_{l+1}} \langle (f(g_{l+1}), n'_l, \alpha_l), ss'_l[t'_{l+1} := f(w)f(w')]\rangle$.

(SEQEND): We have $\langle c, g_l, n_l, ss_l\rangle \rightsquigarrow_{t_{l+1}} \langle c, g_l, n_l, ss_l[t_{l+1} := \epsilon]\rangle$, where $t_{l+1} \in c$, $t_{l+1} \leq n_l$ and $ss_l(t_{l+1}) = \$$. By the induction hypothesis, we have $dom(f) = \{t \in Tid \mid 0 \leq t \leq n_l\}$. Therefore $t_{l+1} \in dom(f) \cap c$ and by the induction hypothesis, we get $f(t_{l+1}) \leq n'_l$. Let $t'_{l+1} = f(t_{l+1})$. By rule (ABSSEQEND), we get $(\langle (g_l, n'_l, \alpha_l), \$\rangle\ ,\ \langle (g_l, n'_l, \alpha_l \cup \{t'_{l+1}\}), \epsilon\rangle) \in \Delta_{t'_{l+1}}$. By the induction hypothesis, we have $ss'_l(t'_{l+1}) = f(ss_l(t_{l+1})) = f(\$) = \$$. Therefore, we have $\langle (g_l, n'_l, \alpha_l), ss'_l\rangle \longrightarrow_{t'_{l+1}} \langle (g_l, n'_l, \alpha_l \cup \{t'_{l+1}\}), ss'_l[t'_{l+1} := \epsilon]\rangle$.

(FORK): We have $\langle c, g_l, n_l, ss_l\rangle \rightsquigarrow_{t_{l+1}} \langle c, g_{l+1}, n_l + 1, ss_l[t_{l+1} := ww']\rangle$, where $t_{l+1} \in c$, $t_{l+1} \leq n_l$, $ss_l(t_{l+1}) = \gamma w'$, and $(n_l+1, \langle g_l, \gamma\rangle, \langle g_{l+1}, w\rangle) \in \Delta_F$. By the induction hypothesis, we have $dom(f) = \{t \in Tid \mid 0 \leq t \leq n_l\}$. Therefore $t_{l+1} \in dom(f) \cap c$ and by the induction hypothesis, we get $f(t_{l+1}) \leq n'_l$. Let $t'_{l+1} = f(t_{l+1})$. There are two cases:

1. ($n_l+1 \in c$): Since $|c| \leq k$, $n_l+1 \notin dom(f)$, and $n_l+1 \in c$, we have $|dom(f) \cap c| < k$. Therefore $|\{t \in Tid \mid 0 \leq t \leq n'_l\}| = |dom(f) \cap c| < k$, which implies that $n'_l + 1 < k$. We extend the renaming function f so that $f(n_l + 1) = n'_l + 1$. By Assumption A2.1, we have $(n'_l + 1, \langle f(g_l), f(\gamma)\rangle, \langle f(g_{l+1}), f(w)\rangle) \in \Delta_F$. By rule (ABSFORK), we get $(\langle (f(g_l), n'_l, \alpha_l), f(\gamma)\rangle\ ,\ \langle (f(g_{l+1}), n'_l + 1, \alpha_l), f(w)\rangle) \in \Delta_{t'_{l+1}}$. By the induction hypothesis, we have $ss'_l(t'_{l+1}) = f(ss_l(t_{l+1})) = f(\gamma)f(w')$. From Lemmas 5 and 6 and the property of γ_0, we get that $ss'_l(f(n_l + 1)) = ss'_l(n'_l + 1) = \gamma_0\$ = f(\gamma_0\$) = f(ss_l(n_1 + 1))$. Therefore, we have $\langle (f(g_l), n'_l, \alpha_l), ss'_l\rangle \longrightarrow_{t'_{l+1}} \langle (f(g_{l+1}), n'_l + 1, \alpha_l), ss'_l[t'_{l+1} := f(w)f(w')]\rangle$.
2. ($n_l + 1 \notin c$): We extend the renaming function f so that $f(n_l + 1) = k$. By Assumption A2.1, we have $(k, \langle f(g_l), f(\gamma)\rangle, \langle f(g_{l+1}), f(w)\rangle) \in \Delta_F$. By the rule (ABSFORKNONDET), we get $(\langle (f(g_l), n'_l, \alpha_l), f(\gamma)\rangle\ ,\ \langle (f(g_{l+1}), n'_l, \alpha_l), f(w)\rangle) \in \Delta_{t'_{l+1}}$. By the induction hypothesis, we have $ss'_l(t'_{l+1}) = f(ss_l(t_{l+1})) = f(\gamma)f(w')$. From Lemmas 5 and 6 and the property of γ_0, we get that $ss'_l(f(n_l + 1)) = ss'_l(k) = \gamma_0\$ = f(\gamma_0\$) = f(ss_l(n_1 + 1))$. Therefore, we have $\langle (f(g_l), n'_l, \alpha_l), ss'_l\rangle \longrightarrow_{t'_{l+1}} \langle (f(g_{l+1}), n'_l, \alpha_l), ss'_l[t'_{l+1} := f(w)f(w')]\rangle$.

(JOIN) We have $\langle c, g_l, n_l, ss_l\rangle \rightsquigarrow_{t_{l+1}} \langle c, g_{l+1}, n_l, ss_l[t_{l+1} := ww']\rangle$, where $t_{l+1} \in c$, $t_{l+1} \leq n_l$, $ss_l(t_{l+1}) = \gamma w'$, $(x, \langle g_l, \gamma\rangle, \langle g_{l+1}, w\rangle) \in \Delta$ for some $x \in LTV$, and $ss_l(\gamma(x)) = \epsilon$.

By the induction hypothesis, we have $dom(f) = \{t \in Tid \mid 0 \leq t \leq n_l\}$. Therefore $t_{l+1} \in dom(f) \cap c$ and by the induction hypothesis, we get $f(t_{l+1}) \leq n'_l$. Let $t'_{l+1} = f(t_{l+1})$. By Assumption A3.1, we have $(x, \langle f(g_l), f(\gamma)\rangle, \langle f(g_{l+1}), f(w)\rangle) \in \Delta$. By the induction hypothesis, we have $ss'_l(f(\gamma)(x)) = ss'_l(f(\gamma(x))) = f(ss_l(\gamma(x))) = f(\epsilon) = \epsilon$. From Lemma 7, we get $f(\gamma)(x) \in \alpha_l$. By rule (ABSJOIN), we get $(\langle (f(g_l), n'_l, \alpha_l), f(\gamma)\rangle\ ,\ \langle (f(g_{l+1}), n'_l, \alpha_l), f(w)\rangle) \in \Delta_{t'_{l+1}}$. By the induction hypothesis, we have $ss'_l(t'_{l+1}) = f(ss_l(t_{l+1})) = f(\gamma)f(w')$. Therefore, we have $\langle (f(g_l), n'_l, \alpha_l), ss'_l\rangle \longrightarrow_{t'_{l+1}} \langle (f(g_{l+1}), n'_l, \alpha_l), ss'_l[t'_{l+1} := f(w)f(w')]\rangle$.

□

Theorem 5 *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 $f : Tid \to Tid_k$ 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 $j \in Tid$.*

PROOF The proof follows immediately from Lemma 8. □

Lemma 9 *For every k-bounded execution $\langle (g'_0, n'_0, \alpha_0), ss'_0\rangle \longrightarrow_{t'_1} \langle (g'_1, n'_1, \alpha_1), ss'_1\rangle \longrightarrow_{t'_2} \ldots \longrightarrow_{t'_l} \langle (g'_l, n'_l, \alpha_l), ss'_l\rangle$ of P_k, there is a renaming function $f : Tid \to Tid_k$ and an execution $\langle g_0, n_0, ss_0\rangle \rightsquigarrow_{t_1} \langle g_1, n_1, ss_1\rangle \rightsquigarrow_{t_2} \ldots \rightsquigarrow_{t_l} \langle g_l, n_l, ss_l\rangle$ of P such that the following statements are true.*

1. *$dom(f) = \{t \in Tid \mid 0 \leq t \leq n_l\}$.*
2. *f maps $dom(f) \setminus \{t \in Tid \mid f(t) = k\}$ one-one onto $\{t \in Tid \mid 0 \leq t \leq n'_l\}$.*
3. *$g'_l = f(g_l)$.*
4. *For all $t \in dom(f)$, $ss'_l(f(t)) = f(ss_l(t))$.*

PROOF We do the proof by induction over the length l of the k-bounded execution sequence of the dynamic concurrent pushdown system.

Base case ($l = 0$): Let f be such that $f(0) = 0$ and f is undefined elsewhere. We know that $n'_0 = 0$ and $n_0 = 0$. Therefore $dom(f) = \{t \in Tid \mid 0 \leq t \leq n_l\}$ and f maps $dom(f)$ one-one onto $\{t \in Tid \mid 0 \leq t \leq n'_l\}$. From the property of g_0, we get that $g'_0 = f(g_0) = g_0$. From the property of γ_0, we get that $\gamma'_0 = f(\gamma_0) = \gamma_0$. Therefore $ss'_0(f(0)) = ss'_0(0) = \gamma_0\$ = ss_0(0)$.

Inductive case: We assume that the theorem holds for l with a renaming function f. We now prove the theorem for $l + 1$ by a case analysis.

(ABSSEQ): We have $\langle (g'_l, n'_l, \alpha_l), ss'_l\rangle \longrightarrow_{t'_{l+1}} \langle (g'_{l+1}, n'_l, \alpha_l), ss'_{l+1}\rangle$, where $0 \leq t'_{l+1} \leq n'_l$. By the induction hypothesis, we have $dom(f) = \{t \in Tid \mid 0 \leq t \leq n_l\}$ and there is $t_{l+1} \in dom(f)$ such that $t'_{l+1} = f(t_{l+1})$. Let $ss_l(t_{l+1}) = \gamma w'$. By the induction hypothesis, we have $ss'_l(t'_{l+1}) = f(ss_l(t_{l+1})) = f(\gamma)f(w')$ and $g'_l = f(g_l)$. Let $ss'_{l+1}(t'_{l+1}) = w'_{l+1}f(w')$. Then $(\langle f(g_l), f(\gamma)\rangle, \langle g'_{l+1}, w'_{l+1}\rangle) \in \Delta$. By Assumption A1.2, there exist g_{l+1} and w_{l+1} such that $g'_{l+1} = f(g_{l+1})$, $w'_{l+1} = f(w_{l+1})$, and $(\langle g_l, \gamma\rangle, \langle g_{l+1}, w_{l+1}\rangle) \in \Delta$. By rule (SEQ), we get $\langle g_l, n_l, ss_l\rangle \rightsquigarrow_{t_{l+1}} \langle g_{l+1}, n_l, ss_l[t_{l+1} := w_{l+1}w']\rangle$.

(ABSSEQEND): We have $\langle (g'_l, n'_l, \alpha_l), ss'_l\rangle \longrightarrow_{t'_{l+1}} \langle (g'_l, n'_l, \alpha_l \cup \{t'_{l+1}\}), ss'_l[t'_{l+1} := \epsilon]\rangle$, where $0 \leq t'_{l+1} \leq n'_l$ and $ss'_l(t'_{l+1}) = \$$. By the induction hypothesis, we have $dom(f) = \{t \in Tid \mid 0 \leq t \leq n_l\}$ and there is $t_{l+1} \in dom(f)$ such that $t'_{l+1} = f(t_{l+1})$. By the induction hypothesis, we have $\$ = ss'_l(t'_{l+1}) = f(ss_l(t_{l+1}))$. Therefore $ss_l(t_{l+1}) = \$$. By rule (SEQEND), we get $\langle g_l, n_l, ss_l\rangle \rightsquigarrow_{t_{l+1}} \langle g_l, n_l, ss_l[t_{l+1} := \epsilon]\rangle$.

(ABSFORK): We have $\langle (g'_l, n'_l, \alpha_l), ss'_l\rangle \longrightarrow_{t'_{l+1}} \langle (g'_{l+1}, n'_l + 1, \alpha_l), ss'_{l+1}\rangle$, where $0 \leq t'_{l+1} \leq n'_l$ and $n'_l + 1 < k$. By the induction hypothesis, we have $dom(f) = \{t \in Tid \mid 0 \leq t \leq n_l\}$ and there is $t_{l+1} \in dom(f)$ such that $t'_{l+1} = f(t_{l+1})$. Let $ss_l(t_{l+1}) = \gamma w'$. By the induction hypothesis, we have $ss'_l(t'_{l+1}) = f(ss_l(t_{l+1})) = f(\gamma)f(w')$ and $g'_l = f(g_l)$. Let $ss'_{l+1}(t'_{l+1}) = w'_{l+1}f(w')$. Extend f to $n_l + 1$ so that $f(n_l + 1) = n'_l + 1$. Then $(f(n_l + 1), \langle f(g_l), f(\gamma)\rangle, \langle g'_{l+1}, w'_{l+1}\rangle) \in \Delta_F$. By Assumption A2.2, there exist g_{l+1} and w_{l+1} such that $g'_{l+1} = f(g_{l+1})$, $w'_{l+1} = f(w_{l+1})$, and $(n_l + 1, \langle g_l, \gamma\rangle, \langle g_{l+1}, w_{l+1}\rangle) \in \Delta_F$. By rule (FORK), we get $(g_l, n_l, ss_l) \rightsquigarrow_{t_{l+1}} (g_{l+1}, n_l + 1, ss_l[t_{l+1} := w_{l+1}w'])$. From Lemmas 5 and 6 and the property of γ_0, we get that $ss'_l(f(n_l + 1)) = ss'_l(n'_l + 1) = \gamma_0\$ = f(\gamma_0\$) = f(ss_l(n_1 + 1))$.

(ABSFORKNONDET): We have $\langle (g'_l, n'_l, \alpha_l), ss'_l\rangle \longrightarrow_{t'_{l+1}} \langle (g'_{l+1}, n'_l, \alpha_l), ss'_{l+1}\rangle$, where $0 \leq t'_{l+1} \leq n'_l$. By the induction hypothesis, we have $dom(f) = \{t \in Tid \mid 0 \leq t \leq n_l\}$ and there is $t_{l+1} \in dom(f)$ such that $t'_{l+1} = f(t_{l+1})$. Let $ss_l(t_{l+1}) = \gamma w'$. By the induction hypothesis, we have $ss'_l(t'_{l+1}) = f(ss_l(t_{l+1})) = f(\gamma)f(w')$

and $g'_l = f(g_l)$. Let $ss'_{l+1}(t'_{l+1}) = w'_{l+1}f(w')$. Extend f to $n_l + 1$ so that $f(n_l + 1) = k$. Then $(f(n_l + 1), \langle f(g_l), f(\gamma)\rangle, \langle g'_{l+1}, w'_{l+1}\rangle) \in \Delta_F$. By Assumption A2.2, there exist g_{l+1} and w_{l+1} such that $g'_{l+1} = f(g_{l+1})$, $w'_{l+1} = f(w_{l+1})$, and $(n_l + 1, \langle g_l, \gamma\rangle, \langle g_{l+1}, w_{l+1}\rangle) \in \Delta_F$. By rule (FORK), we get $\langle g_l, n_l, ss_l\rangle \rightsquigarrow_{t_{l+1}} \langle g_{l+1}, n_l + 1, ss_l[t_{l+1} := w_{l+1}w']\rangle$. From Lemmas 5 and 6 and the property of γ_0, we get that $ss'_l(f(n_l + 1)) = ss'_l(n'_l + 1) = \gamma_0\$ = f(\gamma_0\$) = f(ss_l(n_1 + 1))$.

(ABSJOIN) We have $\langle (g'_l, n'_l, \alpha_l), ss'_l\rangle \longrightarrow_{t'_{l+1}} \langle (g'_{l+1}, n'_l, \alpha_l), ss'_{l+1}\rangle$, where $0 \leq t'_{l+1} \leq n'_l$. By the induction hypothesis, we have $dom(f) = \{t \in Tid \mid 0 \leq t \leq n_l\}$ and there is $t_{l+1} \in dom(f)$ such that $t'_{l+1} = f(t_{l+1})$. Let $ss_l(t_{l+1}) = \gamma w'$. By the induction hypothesis, we have $ss'_l(t'_{l+1}) = f(ss_l(t_{l+1})) = f(\gamma)f(w')$ and $g'_l = f(g_l)$. Let $ss'_{l+1}(t'_{l+1}) = w'_{l+1}f(w')$. Then there is $x \in LTV$ such that $(x, \langle f(g_l), f(\gamma)\rangle, \langle g'_{l+1}, w'_{l+1}\rangle) \in \Delta_J$ and $f(\gamma)(x) \in \alpha_l$. By Assumption A3.2, there exist g_{l+1} and w_{l+1} such that $g'_{l+1} = f(g_{l+1})$, $w'_{l+1} = f(w_{l+1})$, and $(x, \langle g_l, \gamma\rangle, \langle g_{l+1}, w_{l+1}\rangle) \in \Delta_J$. From Lemma 7, we get $ss'_l(f(\gamma)(x)) = \epsilon$. By the induction hypothesis, we get $ss_l(\gamma(x)) = \epsilon$. By rule (JOIN), we get $\langle g_l, n_l, ss_l\rangle \rightsquigarrow_{t_{l+1}} \langle g_{l+1}, n_l, ss_l[t_{l+1} := w_{l+1}w']\rangle$.

□

Theorem 6 *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 $f : Tid \to Tid_k$ 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 $j \in Tid$.*

PROOF The proof follows immediately from Lemma 9. □