Summarizing Procedures in Concurrent Programs

Shaz Qadeer Sriram K. Rajamani Jakob Rehof
Microsoft Research
One Microsoft Way
Redmond, WA 98052

Abstract

The ability to summarize procedures is fundamental to building scalable interprocedural analyses. For sequential programs, procedure summarization is well-understood and used routinely in a variety of compiler optimizations and software defect-detection tools. However, the benefit of summarization is not available to multithreaded programs, for which a clear notion of summaries has so far remained unarticulated in the research literature.

In this paper, we present an intuitive and novel notion of procedure summaries for multithreaded programs. We also present a model checking algorithm for these programs that uses procedure summarization as an essential component. Our algorithm can also be viewed as a precise interprocedural dataflow analysis for multithreaded programs. Our method for procedure summarization is based on the insight that in well-synchronized programs, any computation of a thread can be viewed as a sequence of transactions, each of which appears to execute atomically to other threads. We summarize within each transaction; the summary of a procedure comprises the summaries of all transactions within the procedure. We leverage the theory of reduction [17] to infer boundaries of these transactions.

The procedure summaries computed by our algorithm allow reuse of analysis results across different call sites in a multithreaded program, a benefit that has hitherto been available only to sequential programs. Although our algorithm is not guaranteed to terminate on multithreaded programs that use recursion (reachability analysis for multithreaded programs with recursive procedures is undecidable [18]), there is a large class of programs for which our algorithm does terminate. We give a formal characterization of this class, which includes programs that use shared variables, synchronization, and recursion.

Categories and Subject Descriptors: D.1.3: Concurrent programming, parallel programming; D.2.4: Software/program verification

General Terms: Reliability, security, languages, verification

Keywords: Concurrent programs, pushdown systems, model checking, interprocedural dataflow analysis, procedure summaries, transactions, reduction

POPL'04, January 14–16, 2004, Venice, Italy.

1 Introduction

Sequential programs with finite-domain variables and recursive procedures are infinite-state systems due to unbounded stack depth. In spite of the potentially infinite state space of these programs, assertion checking is decidable for them. A common technique for analyzing such programs is CFL reachability [21, 19] (or equivalently, pushdown model checking [22, 10]), where the key idea is to build procedure summaries. The *summary* of a procedure P contains the state pair (s, s') if in state s, there is an invocation of P that yields the state s' on termination. Summaries enable reuse—if P is called from two different places with the same state s, the work done in analyzing the first call is reused for the second. This reuse is the key to scalability of interprocedural analyses. Additionally, summarization avoids direct representation of the call stack, and guarantees termination of the analysis even if the program has recursion.

Assertion checking for multithreaded programs with finite-domain variables and recursive procedures is undecidable [18]. Most approaches to the analysis of such programs either restrict the interaction between synchronization and procedure calls [1, 9, 6], or perform dataflow-style overapproximations [20], or use abstractions to capture the behavior of each thread [4, 12]. In this paper, we take a radically different approach to solve this problem. We perform model checking directly on the multithreaded program. However, in order to gain reuse and scalability in our analysis, we present a novel method to compute procedure summaries even in the presence of multiple threads. Since the assertion checking problem is undecidable for multithreaded programs with procedures, we cannot guarantee termination of our algorithm on all cases. However, our algorithm terminates for a large class of multithreaded programs that includes programs with shared variables, synchronization, and recursion.

Our model checking algorithm for multithreaded programs has two levels. The first level performs reachability analysis and maintains an explicit stack for each thread. The second level computes a summary for each procedure. During the reachability analysis at the first level, whenever a thread makes a procedure call, we invoke the second level to compute a summary for the procedure. This summary is returned to the first level, which uses it to continue the reachability analysis.

The most crucial aspect of this algorithm is the notion of procedure summaries in multithreaded programs. A straightforward generalization of a (sequential) procedure summary to the case of multithreaded programs could attempt to accumulate all state pairs (s, s') obtained by invoking this procedure in any thread. But this simple-

minded extension is not that meaningful, since the resulting state s' for an invocation of a procedure P in a thread might reflect updates by interleaved actions of concurrently executing threads. Clearly, these interleaved actions may depend on the local states of the other threads. Thus, if (s, s') is an element of such a summary, and the procedure P is invoked again by some thread in state s, there is no guarantee that the invoking thread will be in state s' on completing execution of P.

We present a robust and intuitive notion of procedure summaries for multithreaded programs. This notion is based on the insight that in well-synchronized programs, any computation of a thread can be viewed as a sequence of transactions, each of which appears to execute atomically to other threads. While analyzing the execution of a transaction by a thread, interleavings with other threads need not be considered. Our key idea is to summarize procedures within such transactions. Two main technical difficulties arise while performing transaction-based summarization of procedures:

- Transaction boundaries may not coincide with procedure boundaries. A transaction may begin in a procedure `foo` and end half-way inside a procedure `bar` called from `foo`. Conversely, a transaction may begin in a procedure `foo` and continue even after `foo` returns. One way to summarize such transactions is to have a stack frame as part of the state in each summary. However, this solution not only complicates the algorithm but also makes the summaries unbounded even if all state variables have a finite domain. Our summaries do *not* contain stack frames. If a transaction begins in one procedure context and ends in another procedure context, we break up the summary into smaller sub-summaries each within the context of a single procedure. Thus, our model checking algorithm uses a combination of two representations—states with stacks and summaries without stacks.
- A procedure can be called from different phases of a transaction —the pre-commit phase or the post-commit phase. We need to summarize the procedure differently depending on the phase of the transaction at the call site. We solve this problem by instrumenting the source program with a boolean variable representing the transaction phase, thus making the transaction phase part of the summaries.

We present a formal characterization of a class of multithreaded programs on which our summarization-based model checking algorithm is guaranteed to terminate. We show that if every call to a recursive procedure is contained entirely within a transaction, our algorithm will terminate with the correct answer.

To detect transactions in multithreaded programs, we leverage the theory of reduction [17]. Reduction views a transaction as a sequence of actions $a_1, \ldots, a_m, x, b_1, \ldots, b_n$ such that each a_i is a right mover and each b_i is a left mover. A right mover is an action that commutes to the right of every action by another thread; a left mover is an action that commutes to the left of every action by another thread. Thus, to detect transactions we need to detect right and left movers. In this paper, we abstract away from the problem of detecting movers and instead focus on the algorithm for summarization assuming that right and left movers are provided by a separate analysis [15, 14, 13].

To summarize (!), we present a novel two-level model checking algorithm for multithreaded programs with (recursive) procedures. This algorithm uses transaction-based procedure summarization as a core component. We present a formal characterization of a class of programs on which our algorithm is guaranteed to terminate.

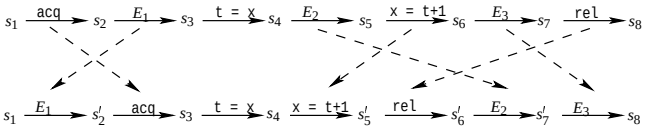


Figure 1. Transaction

This class includes a number of realistic programs that use shared variables, synchronization, and procedures. In recent years, flow-sensitive and context-sensitive analyses have been used to check finite state properties of sequential programs in a scalable way using summarization as an essential component [3, 5]. Our work is a first step in bringing these analyses to bear on multithreaded programs as well.

2 Overview

We illustrate our ideas using examples that use mutexes to protect accesses to shared variables. We first make some general observations about such programs.

- The action `acquire(m)`, where `m` is a mutex, is a right mover. Once it happens, there is no enabled action of another thread that may access `m`. Hence, this action can be commuted to the right of any action of another thread.
- The action `release(m)` is a left mover. At a point when it is enabled but has not happened, there is no enabled action of another thread that may access `m`. Hence, this action can be commuted to the left of any action of another thread.
- An action that accesses only local variables is both a left mover and a right mover, since this action can be commuted both to the left and the right of actions by the other threads.
- An action that accesses a shared variable is both a left mover and right mover, as long as all threads acquire the same mutex before accessing that variable.

A transaction is a sequence of right movers, followed by a single atomic action with no restrictions, followed by a sequence of left movers. A transaction can be in two states: pre-commit or post-commit. A transactions starts in the pre-commit state and stays in the pre-commit state as long as right movers are being executed. When the atomic action (with no restrictions) is executed, the transaction moves to the post-commit state. This atomic action is called the *committing action*. The transaction stays in the post-commit state as long as left movers are being executed until the transaction completes.

We illustrate a transaction with an example. Figure 1 shows an execution of a concurrent program. A thread performs the following sequence of four operations: (1) acquires a lock (the operation `acq` in the first execution trace), (2) reads a variable `x` protected by that lock into a local variable `t` (`t=x`), (3) updates that variable (`x=t+1`), and (4) releases the lock (`rel`). Suppose that the actions of this method are interleaved with arbitrary actions E_1, E_2, E_3 of other threads. We assume that the environment actions respect the locking discipline of accessing `x` only after acquiring the lock.

Since the acquire operation is a right mover, it is commuted to the right of the environment action E_1 without changing the final state s_3, even though the intermediate state changes from s_2 to s'_2. The read operation is the committing action. The write and release operations are left movers and are commuted to the left of environment

```
bool available[N];
mutex m;

int getResource() {
      int i = 0;
L0:   acquire(m);
L1:   while (i < N) {
L2:     if (available[i]) {
L3:       available[i] = false;
L4:       release(m);
L5:       return i;
        }
L6:     i++;
      }
L7:   release(m);
L8:   return i;
}
```

Figure 2. Resource allocator with coarse-grained locking

```
bool available[N];
mutex m[N];

int getResource() {
      int i = 0;
L0:   while (i < N) {
L1:     acquire(m[i]);
L2:     if (available[i]) {
L3:       available[i] = false;
L4:       release(m[i]);
L5:       return i;
        } else {
L6:       release(m[i]);
        }
L7:     i++;
      }
L8:   return i;
}
```

Figure 3. Resource allocator with fine-grained locking

actions E_2 and E_3. Finally, after performing a series of commute operations, we get the execution at the bottom of the diagram in which the actions of the first thread happen without interruption from the environment. Note that in this execution the initial and final states are the same as before. Thus, the sequence of operations performed by the first thread constitute a transaction. This transaction is in the pre-commit state before the read of `x` and in the post-commit state afterwards.

We illustrate different aspects of our summarization-based model checking algorithm using four examples. The first example illustrates a simple case where the transaction boundary and the procedure boundary coincide. The second example illustrates summarization where a procedure body contains several transactions inside it. The third example illustrates that our algorithm terminates and validates the program assertions even in the presence of unbounded recursion. The fourth example illustrates our summarization technique when a procedure is called from different transactional contexts.

Example 1

Consider the resource allocation routine shown in Figure 2. There are `N` shared resources numbered $0, \ldots, \mathtt{N}-1$. The j-th entry in the global boolean array `available` is true iff the j-th resource is free to be allocated. A mutex `m` is used to protect accesses to `available`. The mutex `m` has the value 0 when free, and 1 when locked. The body of `getResource` acquires the mutex `m`, and then searches for the first available resource. If a free resource is found at index `i`, it sets `available[i]` to `false` and releases the mutex. If no free resource is found, then it releases the mutex. In both cases, it returns the final value of `i`. Thus, the returned value is the index of a free resource if one is available; otherwise, it is `N`. There is a companion procedure `freeResource` for freeing an allocated resource, but we have not shown that in the figure. We assume that the multithreaded program consists of a number of threads that nondeterministically call `getResource` and `freeResource`.

Since `acquire(m)` is a right mover, `release(m)` is a left mover, and all other actions in `getResource` are both right and left movers, the entire procedure is contained in a single transaction. Suppose `N` = 2. We use $\langle a_0, a_1 \rangle$ to denote the contents of `available`, where a_0 and a_1 respectively denote the values of `available[0]` and `available[1]`. The summary of `getResource` consists of a set of edges of the form $(pc, i, m, \langle a_0, a_1 \rangle) \mapsto (pc', i', m', \langle a_0', a_1' \rangle)$, where the tuple $(pc, i, m, \langle a_0, a_1 \rangle)$ represents the values of the program counter and variables `i`, `m`, and `available` in the pre-store of the transaction and the tuple $(pc', i', m', \langle a_0', a_1' \rangle)$ denotes the corresponding values in the post-store of the transaction. The computed summary of `getResource` consists of the following edges:

$$\begin{array}{lcl}
(\mathtt{L0}, 0, 0, \langle 0, 0 \rangle) & \mapsto & (\mathtt{L8}, 2, 0, \langle 0, 0 \rangle) \\
(\mathtt{L0}, 0, 0, \langle 0, 1 \rangle) & \mapsto & (\mathtt{L5}, 1, 0, \langle 0, 0 \rangle) \\
(\mathtt{L0}, 0, 0, \langle 1, 0 \rangle) & \mapsto & (\mathtt{L5}, 0, 0, \langle 0, 0 \rangle) \\
(\mathtt{L0}, 0, 0, \langle 1, 1 \rangle) & \mapsto & (\mathtt{L5}, 0, 0, \langle 0, 1 \rangle)
\end{array}$$

All edges in this summary begin at the label `L0` and terminate at one of the two labels—`L8` or `L5`—both of which are labels at which `getResource` returns. Thus, the summary matches the intuition that the procedure body is just one transaction. The first edge summarizes the case when no resource is allocated; the remaining three edges summarize the case when some resource is allocated. There is no edge beginning in a state with $\mathtt{m} = 1$ since from such a state, the execution of the transaction blocks. Once this summary has been computed, if any thread calls `getResource`, the summary can be used to compute the state at the end of the transaction, without re-analyzing the body of `getResource`, thus providing reuse and scalability.

Example 2

Let us consider a modification to the resource allocator. The new program is shown in Figure 3. We have made the locking more fine-grained by using an array `m` of mutexes and protecting the j-th entry in `available` with the j-th entry in `m`. Now, the body of the procedure `getResource` is no longer contained entirely in a single transaction. In fact, there is one transaction corresponding to each iteration of the loop inside it. Again, suppose `N` = 2. Now, the summary contains edges of the form $(pc, i, \langle m_0, m_1 \rangle, \langle a_0, a_1 \rangle) \mapsto (pc', i', \langle m_0', m_1' \rangle, \langle a_0', a_1' \rangle)$, where $\langle m_0, m_1 \rangle$ denotes the contents of `m` in the pre-store and $\langle m_0', m_1' \rangle$ denotes the contents of `m` in the post-store. The computed summary of `getResource` consists of the following edges:

$$\begin{array}{lcl}
(\mathtt{L0}, 0, \langle 0, 0 \rangle, \langle 0, 0 \rangle) & \mapsto & (\mathtt{L1}, 1, \langle 0, 0 \rangle, \langle 0, 0 \rangle) \\
(\mathtt{L0}, 0, \langle 0, 0 \rangle, \langle 0, 1 \rangle) & \mapsto & (\mathtt{L1}, 1, \langle 0, 0 \rangle, \langle 0, 1 \rangle) \\
(\mathtt{L0}, 0, \langle 0, 0 \rangle, \langle 1, 0 \rangle) & \mapsto & (\mathtt{L5}, 0, \langle 0, 0 \rangle, \langle 0, 0 \rangle) \\
(\mathtt{L0}, 0, \langle 0, 0 \rangle, \langle 1, 1 \rangle) & \mapsto & (\mathtt{L5}, 0, \langle 0, 0 \rangle, \langle 0, 1 \rangle)
\end{array}$$

```
int g = 0;
mutex m;
void foo(int r) {          void main() {
L0:   if (r == 0) {              int q = choose({0,1});
L1:      foo(r);           M0:   foo(q);
      } else {             M1:   acquire(m)
L2:      acquire(m);       M2:   assert(g >= 1);
L3:      g++;              M3:   release(m);
L4:      release(m);       M4:   return;
      }                    }
L5:   return;
}

P = { main() } || { main() }
```

Figure 4. Summarization enables termination

$$
\begin{array}{rcl}
(\texttt{L1}, 1, \langle 0,0\rangle, \langle 0,0\rangle) & \mapsto & (\texttt{L8}, 2, \langle 0,0\rangle, \langle 0,0\rangle) \\
(\texttt{L1}, 1, \langle 0,0\rangle, \langle 0,1\rangle) & \mapsto & (\texttt{L5}, 1, \langle 0,0\rangle, \langle 0,0\rangle) \\
(\texttt{L1}, 1, \langle 0,0\rangle, \langle 1,0\rangle) & \mapsto & (\texttt{L8}, 2, \langle 0,0\rangle, \langle 1,0\rangle) \\
(\texttt{L1}, 1, \langle 0,0\rangle, \langle 1,1\rangle) & \mapsto & (\texttt{L5}, 1, \langle 0,0\rangle, \langle 1,0\rangle)
\end{array}
$$

This summary contains three kinds of edges. The first two edges correspond to the transaction that starts at the beginning of the procedure, i.e., at label `L0` with $\texttt{i} = 0$, goes through the loop once, and ends at `L1` with $\texttt{i} = 1$. The next two edges correspond to the transaction that again starts at the beginning of the procedure, but ends with the return at label `L5` during the first iteration through the loop. The last four edges correspond to the transaction that starts in the middle of the procedure at label `L1` with $\texttt{i} = 1$, and returns either at label `L5` or `L8`. Note that edges of the first and third kind did not exist in the summary of the previous version of `getResource`, where all edges went from entry to exit.

Example 3

The previous two examples illustrated summaries and the reuse afforded by them. In a number of programs, summaries enable our model checking algorithm to terminate where the naive model checking algorithm without summaries will not terminate. Consider the example in Figure 4. The program `P` consists of two threads, each of which starts execution by calling the `main` procedure. The `main` procedure has a local variable `q` which is initialized nondeterministically. Then `main` calls `foo` with `q` as the actual parameter. The procedure `foo` has an infinite recursion if the parameter `r` is `0`. Otherwise, it increments global `g` and returns. After returning from `foo`, the `main` procedure asserts that `(g >= 1)`. All accesses to the shared global `g` are protected by a mutex `m`. The initial value of `g` is `0`.

The stack can grow without bound due to the recursion in procedure `foo`. Hence, naive model checking does not terminate on this example. However, the body of `foo` consists of one transaction, since all action sequences in `foo` consist of a sequence of right movers followed by a sequence of left movers. A summary edge for `foo` is of the form $(pc, r, m, g) \mapsto (pc', r', m', g')$, whose meaning is similar to that of a summary edge in the previous examples. The summary for `foo` consists of the following edges:

$$
\begin{array}{rcl}
(\texttt{L0}, 1, 0, 0) & \mapsto & (\texttt{L5}, 1, 0, 1) \\
(\texttt{L0}, 1, 0, 1) & \mapsto & (\texttt{L5}, 1, 0, 2)
\end{array}
$$

There is no edge beginning in a state with $\texttt{r} = 0$ since from such a state, the execution of the transaction never terminates. Using summaries, we avoid reasoning about the stack explicitly inside `foo` and also avoid exploring the unbounded recursion in `foo`.

```
int gm = 0, gn = 0;
mutex m, n;
void bar() {
N0: acquire(m);
N1: gm++;
N2: release(m);
}

 void foo1() {              void foo2() {
 L0:     acquire(n);        M0:     acquire(n);
 L1:     gn++;              M1:     gn++;
 L2:     bar();             M2:     release(n);
 L3:     release(n);        M3:     bar();
 L4:     return;            M4:     return;
 }                          }

P = { foo1() } || { foo2() }
```

Figure 5. Summarization from different transactional contexts

The body of `main` has two transactions. The first transaction begins at label `M0` and ends at label `M1`, consisting of essentially the call to `foo`. The second transaction begins at label `M1` and ends at label `M4`. A summary edge for `main` has the form $(pc, q, m, g) \mapsto (pc, q', m', g')$. The summary for `main` consists of the following edges:

$$
\begin{array}{rcl}
(\texttt{M0}, 1, 0, 0) & \mapsto & (\texttt{M1}, 1, 0, 1) \\
(\texttt{M0}, 1, 0, 1) & \mapsto & (\texttt{M1}, 1, 0, 2)
\end{array}
$$

$$
\begin{array}{rcl}
(\texttt{M1}, 1, 0, 1) & \mapsto & (\texttt{M4}, 1, 0, 1) \\
(\texttt{M1}, 1, 0, 2) & \mapsto & (\texttt{M4}, 1, 0, 2)
\end{array}
$$

Using these summaries for procedures `foo` and `main`, our model checking algorithm is able to terminate and correctly conclude that P is free of assertion violations. The algorithm begins with an empty stack for each thread. When a thread calls `main`, since the body of `main` is not contained within one transaction, the algorithm pushes a frame for `main` on the stack of the calling thread. However, when a thread calls `foo`, no frame corresponding to `foo` is pushed since the entire body of `foo` is contained within a transaction. Instead, `foo` is summarized and its summary is used to make progress in the model checking.

Example 4

Consider the example in Figure 5. Here, two shared variables `gm` and `gn` are protected by mutexes `m` and `n` respectively. Procedure `bar` accesses the variable `gm`, and is called from two different procedures `foo1` and `foo2`. In `foo1`, the procedure `bar` is called from the pre-commit state of the transaction, since no mutexes are released prior to calling `bar`. In `foo2`, the procedure `bar` is called from the post-commit state of the transaction, since mutex `n` is released prior to calling `bar`. The summary for `bar` needs to distinguish these two calling contexts. In the case of the call from `foo1`, the entire body of `foo1` including the call to `bar()` is part of the same transaction. In the case of the call from `foo2`, there are two transactions, one from label `M0` to `M3`, and another from label `M3` to `M4`. We distinguish these two by instrumenting the semantics of the program with an extra bit of information that records the phase of the transaction. Then, each summary edge provides the pre and post values not only of program variables but also of the transaction phase. More details are given Section 4.

3 Multithreaded programs

The store of a multithreaded program is partitioned into the global store *Global* and the local store *Local* of each thread. The set *Local* of local stores has a special store called *wrong*. The local store of a thread moves to *wrong* on failing an assertion and thereafter the failed thread does not make any other transitions.

Domains

$$\begin{array}{rcrcl} t,u & \in & \mathit{Tid} & = & \{1,\ldots,n\} \\ g & \in & \mathit{Global} & & \\ l & \in & \mathit{Local} & & \\ ls & \in & \mathit{Locals} & = & \mathit{Tid} \to \mathit{Local} \\ f & \in & \mathit{Frame} & & \\ s & \in & \mathit{Stack} & = & \mathit{Frame}^* \\ ss & \in & \mathit{Stacks} & = & \mathit{Tid} \to \mathit{Stack} \\ & & \mathit{State} & = & \mathit{Global} \times \mathit{Locals} \times \mathit{Stacks} \end{array}$$

A multithreaded program (g_0, ls_0, T, T^+, T^-) consists of five components. g_0 is the initial value of the global store. ls_0 maps each thread id $t \in \mathit{Tid}$ to the initial local store $ls_0(t)$ of thread t. We model the behavior of the individual threads using three relations:

$$\begin{array}{rcl} T & \subseteq & \mathit{Tid} \times (\mathit{Global} \times \mathit{Local}) \times (\mathit{Global} \times \mathit{Local}) \\ T^+ & \subseteq & \mathit{Tid} \times \mathit{Local} \times (\mathit{Local} \times \mathit{Frame}) \\ T^- & \subseteq & \mathit{Tid} \times (\mathit{Local} \times \mathit{Frame}) \times \mathit{Local} \end{array}$$

The relation T models thread steps that do not manipulate the stack. The relation $T(t,g,l,g',l')$ holds if thread t can take a step from a state with global store g and local store l, yielding (possibly modified) stores g' and l'. The stack is not accessed or updated during this step. The relation $T^+(t,l,l',f)$ models steps of thread t that push a frame onto the stack. This step does not access the global store, is enabled when the local store is l, updates the local store to l', and pushes the frame f onto the stack. Similarly, the relation $T^-(t,l,f,l')$ models steps of thread t that pop a frame from the stack. This step also does not access the global store, is enabled when the local store is l and the frame at the top of the stack is f, updates the local store to l', and pops the frame f from the stack.

The program starts execution from the state (g_0, ls_0, ss_0) where $ss_0(t) = \varepsilon$ for all $t \in \mathit{Tid}$. At each step, any thread may make a transition. The transition relation $\to_t \subseteq \mathit{State} \times \mathit{State}$ of thread t is defined below. For any function h from A to B, $a \in A$ and $b \in B$, we write $h[a := b]$ to denote a new function such that $h[a := b](x)$ evaluates to $h(x)$ if $x \neq a$, and to b if $x = a$.

Transition relation $\to_t$

$$\frac{T(t,g,ls(t),g',l')}{(g,ls,ss) \to_t (g', ls[t := l'], ss)}$$

$$\frac{T^+(t,ls(t),l',f)}{(g,ls,ss) \to_t (g, ls[t := l'], ss[t := ss(t).f])}$$

$$\frac{ss(t) = s.f \quad T^-(t,ls(t),f,l')}{(g,ls,ss) \to_t (g, ls[t := l'], ss[t := s])}$$

The transition relation $\to \subseteq \mathit{State} \times \mathit{State}$ of the program is the disjunction of the transition relations of the various threads.

$$\to \quad = \quad \exists t.\ \to_t$$

4 Model checking with reduction

Transactions occur in multithreaded programs because of the presence of right and left movers. Inferring which actions of a program are right and left movers is a problem that is important but orthogonal to the contribution of this paper. In this section, we assume that right and left movers are available to us as the result of a previous analysis (see, e.g. [15, 14]). We use this information to derive a model checking algorithm that uses transactions but does not perform any summarization of procedures. We will use the intuition developed in this section to derive a second model checking algorithm in Section 5 that performs procedure summarization as well.

Let $RM, LM \subseteq T$ be subsets of the transition relation T with the following two properties for all $t \neq u$:

1. If $RM(t,g_1,l_1,g_2,l_2)$ and $T(u,g_2,l_3,g_3,l_4)$, there is g_4 such that $T(u,g_1,l_3,g_4,l_4)$ and $RM(t,g_4,l_1,g_3,l_2)$. Further, $RM(u,g_2,l_3,g_3,l_4)$ iff $RM(u,g_1,l_3,g_4,l_4)$, and $LM(u,g_2,l_3,g_3,l_4)$ iff $LM(u,g_1,l_3,g_4,l_4)$.
2. If $T(u,g_1,l_1,g_2,l_2)$ and $LM(t,g_2,l_3,g_3,l_4)$, there is g_4 such that $LM(t,g_1,l_3,g_4,l_4)$ and $T(u,g_4,l_1,g_3,l_2)$. Further, $RM(u,g_1,l_1,g_2,l_2)$ iff $RM(u,g_4,l_1,g_3,l_2)$, and $LM(u,g_1,l_1,g_2,l_2)$ iff $LM(u,g_4,l_1,g_3,l_2)$.

The first property states that a right mover action in thread t commutes to the right of an action of a different thread u. Moreover, the action by thread u is a right mover (resp. left mover) before the commute operation iff it is a right mover (resp. left mover) after the commute operation. Similarly, the second property states the requirement on a left mover in thread t. Our analysis is parameterized by the values of *RM* and *LM* and only requires that they satisfy these two properties. The larger the relations *RM* and *LM*, the longer the transactions our analysis infers. Therefore, these relations should be as large as possible in practice.

As mentioned before, a transaction is a sequence of right movers followed by a single action followed by a sequence of left movers. In order to minimize the number of explored interleavings and to maximize reuse, we would like to infer transactions that are as long as possible. In order to implement this inference, we introduce in each thread a boolean local variable to keep track of the phase of that thread's transaction. Note that this instrumentation is done automatically by our analysis, and not by the programmer. The phase variable of thread t is true if thread t is in the right mover (or pre-commit) part of the transaction; otherwise the phase variable is false. We say that the transaction *commits* when the phase variable moves from true to false. The initial value of the phase variable for each thread is *true*.

$$\begin{array}{rcrcl} p, p' & \in & \mathit{Boolean} & & \\ \ell, \ell' & \in & \mathit{Local}^\# & = & \mathit{Local} \times \mathit{Boolean} \\ \ell s, \ell s' & \in & \mathit{Locals}^\# & = & \mathit{Tid} \to \mathit{Local}^\# \end{array}$$

The initial value of the global store of the instrumented program remains g_0. The initial value of the local stores changes to ℓs_0, where $\ell s_0(t) = \langle ls_0(t), \mathit{true} \rangle$ for all $t \in \mathit{Tid}$. We instrument the transition relation T, T^+, and T^- to generate new transition relations U, U^+, and U^- that update the phase appropriately.

$$\begin{array}{rcl} U & \subseteq & \mathit{Tid} \times (\mathit{Global} \times \mathit{Local}^\#) \times (\mathit{Global} \times \mathit{Local}^\#) \\ U^+ & \subseteq & \mathit{Tid} \times \mathit{Local}^\# \times (\mathit{Local}^\# \times \mathit{Frame}) \\ U^- & \subseteq & \mathit{Tid} \times (\mathit{Local}^\# \times \mathit{Frame}) \times \mathit{Local}^\# \end{array}$$

Ruleset 1: Model checking with reduction

(INIT)

$$\frac{}{\Sigma(g_0, \ell s_0, ss_0)}$$

(STEP)

$$\frac{\forall u \neq t.\ \mathcal{N}(u,g,\ell s(u)) \quad \Sigma(g,ls,ss) \quad U(t,g,\ell s(t),g',\ell')}{\Sigma(g', \ell s[t := \ell'], ss)}$$

(PUSH)

$$\frac{\forall u \neq t.\ \mathcal{N}(u,g,\ell s(u)) \quad \Sigma(g,\ell s,ss) \quad U^+(t,\ell s(t),\ell',f)}{\Sigma(g,\ell s[t := \ell'], ss[t := ss(t).f])}$$

(POP)

$$\frac{\forall u \neq t.\ \mathcal{N}(u,g,\ell s(u)) \quad \Sigma(g,\ell s,ss) \quad U^-(t,\ell s(t),f,\ell') \quad ss(t) = s.f}{\Sigma(g,\ell s[t := \ell'], ss[t := s])}$$

$$\begin{array}{l} U(t,g,\langle l,p\rangle,g',\langle l',p'\rangle) \stackrel{\text{def}}{=} \\ \quad \wedge \quad T(t,g,l,g',l') \\ \quad \wedge \quad p' = (RM(t,g,l,g',l') \wedge (p \vee \neg LM(t,g,l,g',l'))) \end{array}$$

$$\begin{array}{l} U^+(t,\langle l,p\rangle,\langle l',p'\rangle,f) \stackrel{\text{def}}{=} \\ \quad \wedge \quad T^+(t,l,l',f) \\ \quad \wedge \quad p' = p \end{array}$$

$$\begin{array}{l} U^-(t,\langle l,p\rangle,f,\langle l',p'\rangle) \stackrel{\text{def}}{=} \\ \quad \wedge \quad T^-(t,l,f,l') \\ \quad \wedge \quad p' = p \end{array}$$

In the definition of U, the relation between p' and p reflects the intuition that if p is true, then p' continues to be true as long as it executes right mover actions. The phase changes to false as soon as the thread executes an action that is not a right mover. Thereafter, it remains false as long as the thread executes left movers. Then, it becomes true again as soon as the thread executes an action that is a right mover and not a left mover.

For each thread t, we define three sets:

$$\mathcal{R}(t), \mathcal{L}(t), \mathcal{N}(t) \quad \subseteq \quad Global \times Local^{\#}$$

These sets respectively define when a thread is executing in the right mover part of a transaction, the left mover part of a transaction, and outside any transaction. For example, in the first execution of Figure 1, let t be the identifier of the thread executing the transaction. Then, the states $\{s_2, s_3\} \in \mathcal{R}(t)$, $\{s_4, s_5, s_6, s_7\} \in \mathcal{L}(t)$, and $\{s_1, s_8\} \in \mathcal{N}(t)$. These three sets can be any *partition* of $(Global \times Local^{\#})$ satisfying the following two conditions:

C1. $\mathcal{R}(t) \subseteq \{\ (g,\langle l,p\rangle) \mid l \notin \{ls_0(t), wrong\} \wedge p\ \}$.

C2. $\mathcal{L}(t) \subseteq \left\{\ (g,\langle l,p\rangle) \;\middle|\; \begin{array}{l} l \notin \{ls_0(t), wrong\} \wedge \neg p \wedge \\ \forall g', l'.\ T(t,g,l,g',l') \\ \qquad \Rightarrow LM(t,g,l,g',l') \end{array} \right\}$.

Condition C1 says that thread t is in the right mover part of a transaction only if the local store of t is neither its initial value nor *wrong* and the phase variable is true. Condition C2 says that thread t is in the left mover part of a transaction only if the local store of t is neither its initial value nor *wrong*, the phase variable is false, and all possible enabled transitions are left movers. Since $(\mathcal{R}(t), \mathcal{L}(t), \mathcal{N}(t))$ is a partition of $(Global \times Local^{\#})$, once $\mathcal{R}(t)$ and $\mathcal{L}(t)$ have been picked according to C1 and C2, the set $\mathcal{N}(t)$ is implicitly defined.

We write $\mathcal{R}(t,g,\ell)$ whenever $(g,\ell) \in \mathcal{R}(t)$, $\mathcal{L}(t,g,\ell)$ whenever $(g,\ell) \in \mathcal{L}(t)$, and $\mathcal{N}(t,g,\ell)$ whenever $(g,\ell) \in \mathcal{N}(t)$. Finally, using the values of $\mathcal{N}(t)$ for all $t \in Tid$, we model check the multithreaded program by computing the least fixpoint of the set of rules in Ruleset 1. This model checking algorithm schedules a thread only when no other thread is executing inside a transaction.

Conditions C1 and C2 are not quite enough for the model checking algorithm to be sound. The reason is the following. If a transaction in thread t commits but never finishes, the shared variables modified by this transaction become visible to other threads. However, the algorithm does not explore transitions of other threads from any state after the transaction commits. Therefore, we add a third condition C3 which states that every committed transaction must finish. In order to state this condition formally, we extend the transition relation $\rightarrow_t$ from Section 3 to the program store augmented with the phase variable in the natural way.

C3. Suppose $(g,\ell) \in \mathcal{L}(t)$, $\Sigma(g,\ell s,ss)$, and $\ell s(t) = \ell$. Then, there is g', $\ell s'$, and ss' such that $(g', \ell s'(t)) \in \mathcal{N}(t)$ and $(g,\ell s,ss) \rightarrow_t^* (g', \ell s', ss')$.

Our analysis is correct for any partition $(\mathcal{R}(t), \mathcal{L}(t), \mathcal{N}(t))$ of $(Global \times Local^{\#})$ satisfying conditions C1, C2, and C3. The smaller the value of $\mathcal{N}(t)$, the larger the transactions inferred by the analysis. Therefore, an implementation of our analysis should pick a value for $\mathcal{N}(t)$ that is as small as possible. We can now state our soundness theorem for the model checking algorithm presented above.

THEOREM 1. *Let $(g_0, \ell s_0, U, U^+, U^-)$ be the instrumented multithreaded program. Let Σ be the least fixpoint of the rules in Ruleset 1. Let the conditions C1, C2, and C3 be satisfied. If $(g_0, ls_0, ss_0) \rightarrow^* (g, ls, ss)$ and $ls(t) = wrong$, then there is $(g', \ell s', ss')$ and p such that $\Sigma(g', \ell s', ss')$ and $\ell s'(t) = \langle wrong, p\rangle$.*

Proof (Sketch) Suppose $(g_0, ls_0, ss_0) \rightarrow^* (g, ls, ss)$ through some sequence of actions of various threads and $ls(t) = wrong$. First, we extend this sequence to complete all committed but unfinished transactions using condition C3. Then, one by one, we commute each action in an uncommitted transaction to the right and drop it. Eventually, we will get an execution sequence σ with only completed transactions and with the property that σ goes wrong if the original sequence goes wrong. Therefore σ goes wrong as well. In σ, the transactions of a thread could have interleaved actions of another thread. The order in which transactions commit is a total order on the transactions in σ. We denote this total order by $<$. We can transform σ into an equivalent execution σ' (by appropriately right-commuting right movers, and left-commuting left movers), such that σ' has the following properties: (1) for every thread t, no action of a different thread t' occurs in the middle of a transaction of thread t, (2) the transactions in σ' commit in the order $<$. From the properties of right and left movers, we get that σ' also goes wrong. Since σ' schedules each transaction to completion, the states along σ' will be explored by the rules in Ruleset 1. A similar proof has been carried out with more detail in an earlier paper [15]. □

Although this algorithm is sound, it might not terminate if a thread calls a recursive procedure (even if all variables take values from a finite domain). In the next section, we use the concepts developed in this section to derive a model checking algorithm that uses proce-

Ruleset 2: Level I—Reachability

(INIT)

$$\frac{}{\Omega(g_0, \ell s_0, ss_0)}$$

(STEP)

$$\frac{\Omega(g, \ell s, ss) \quad \forall u \neq t.\ \mathcal{N}(u, g, \ell s(u)) \quad Sum(t, g, \ell s(t), g', \ell')}{\Omega(g', \ell s[t := \ell'], ss, ps[t := p'])}$$

(PUSH)

$$\frac{\Omega(g, \ell s, ss) \quad \forall u \neq t.\ \mathcal{N}(u, g, \ell s(u)) \quad Sum^{+}(t, g, \ell s(t), g', \ell', f)}{\Omega(g', \ell s[t := \ell'], ss[t := ss(t).f])}$$

(POP)

$$\frac{\Omega(g, \ell s, ss) \quad \forall u \neq t.\ \mathcal{N}(u, g, \ell s(u)) \quad ss(t) = s.f \quad Sum^{-}(t, g, \ell s(t), f, g', \ell')}{\Omega(g', \ell s[t := \ell'], ss[t := s])}$$

(CFL START)

$$\frac{\Omega(g, \ell s, ss) \quad \forall u \neq t.\ \mathcal{N}(u, g, \ell s(u))}{P(t, g, \ell s(t), g, \ell s(t))}$$

dure summarization and may thus terminate in a lot of cases where the algorithm of this section might not.

5 Model checking with summarization

In this section, we describe our two-level model checking algorithm. The algorithm operates on the instrumented multi-threaded program defined in Section 4. It also uses the partitions $\{(\mathcal{R}(t), \mathcal{L}(t), \mathcal{N}(t)) \mid t \in Tid\}$ defined in Section 4. The algorithm maintains the following relations for performing summarization.

Relations

$$\begin{array}{rcll}
P & \subseteq & Tid & \times(Global \times Local^{\#}) \\
 & & & \times(Global \times Local^{\#}) \\
Sum & \subseteq & Tid & \times(Global \times Local^{\#}) \\
 & & & \times(Global \times Local^{\#}) \\
Sum^{+} & \subseteq & Tid & \times(Global \times Local^{\#}) \\
 & & & \times(Global \times Local^{\#}) \\
 & & & \times Frame \\
Sum^{-} & \subseteq & Tid & \times(Global \times Local^{\#}) \\
 & & & \times Frame \\
 & & & \times(Global \times Local^{\#}) \\
Mark & \subseteq & Tid & \times(Global \times Local^{\#})
\end{array}$$

5.1 Algorithm

Our model checking algorithm operates in two levels. The first-level reachability algorithm is similar to the algorithm in the previous section and maintains a set of reachable states Ω. But it does not use U, U^{+} and U^{-} directly. Instead, it calls into a second-level summarization algorithm that uses U, U^{+} and U^{-} to compute four relations—P, Sum, Sum^{+} and Sum^{-}. Of these four relations, the last three play roles similar to U, U^{+} and U^{-} and are used to communicate results back to the first-level algorithm. Ruleset 2 gives the rules for the first-level reachability algorithm and Ruleset 3 gives the rules for the second-level summarization algorithm.

Ruleset 3: Level II—Summarization

(CFL STEP)

$$\frac{P(t, g_1, \ell_1, g_2, \ell_2) \quad U(t, g_2, \ell_2, g_3, \ell_3) \quad \neg\mathcal{N}(t, g_2, \ell_2)}{P(t, g_1, \ell_1, g_3, \ell_3)}$$

(CFL PUSH)

$$\frac{P(t, g_1, \ell_1, g_2, \ell_2) \quad U^{+}(t, \ell_2, \ell_3, f) \quad \neg\mathcal{N}(t, g_2, \ell_2)}{P(t, g_2, \ell_3, g_2, \ell_3)}$$

(CFL POP)

$$\frac{P(t, g_1, \ell_1, g_2, \ell_2) \quad U^{+}(t, \ell_2, \ell_3, f) \quad \neg\mathcal{N}(t, g_2, \ell_2) \quad Sum^{-}(t, g_2, \ell_3, f, g_3, \ell_4)}{P(t, g_1, \ell_1, g_3, \ell_4)}$$

(CFL SUM^{-})

$$\frac{P(t, g_1, \ell_1, g_2, \ell_2) \quad U^{-}(t, \ell_2, f, \ell_3) \quad \neg\mathcal{N}(t, g_2, \ell_2)}{Sum^{-}(t, g_1, \ell_1, f, g_2, \ell_3)}$$

(CFL SUM)

$$\frac{P(t, g_1, \ell_1, g_2, \ell_2) \quad \mathcal{N}(t, g_2, \ell_2)}{Sum(t, g_1, \ell_1, g_2, \ell_2) \quad Mark(t, g_1, \ell_1)}$$

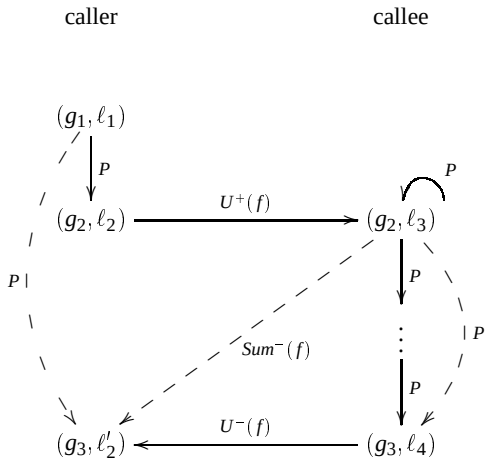
(CFL SUM^{+})

$$\frac{P(t, g_1, \ell_1, g_2, \ell_2) \quad U^{+}(t, \ell_2, \ell_3, f) \quad \neg\mathcal{N}(t, g_2, \ell_2) \quad Mark(t, g_2, \ell_3)}{Sum^{+}(t, g_1, \ell_1, g_2, \ell_3, f) \quad Mark(t, g_1, \ell_1)}$$

Let us refer to elements of $(Global \times Local^{\#})$ as *nodes*. Then, the relations P, Sum, Sum^{+} and Sum^{-} are all *edges* since they connect a pair of nodes. The relation $Mark$ is a subset of nodes. We refer to the relations Sum, Sum^{+} and Sum^{-} as summary edges. These summary edges are computed by the summarization rules (Ruleset 3). The reachability rules and the summarization rules communicate with each other in the following way. The rule (CFL START) creates an edge in P for a thread t when every other thread is outside a transaction. Once summarization has been initiated via (CFL START) from the first level, it continues for as long as a transaction lasts, that is, until the condition $\mathcal{N}(t, g_2, \ell_2)$ becomes true of a target state (g_2, ℓ_2). The summary edges, Sum, Sum^{+}, and Sum^{-}, generated by summarization are used by the reachability rules to do model checking, via the rules STEP, PUSH and POP in Ruleset 2.

The edges in P correspond to both "path edges" and "summary edges" in the CFL reachability algorithm for single-threaded programs [19]. The rule (CFL STEP) is used to propagate path edges within a single procedure, and the rules, (CFL PUSH) and (CFL POP) are used to propagate path edges across procedure boundaries. These rules have analogs in the CFL reachability algorithm.

Figure 6 and Figure 7 illustrate how the rules work in two situations involving function calls. In these figures we assume a fixed thread identifier t, and nodes of the form (g, ℓ) describe the global and local stores of thread t. A path edge $(g, \ell) \xrightarrow{P} (g', \ell')$ indicates that $P(t, g, \ell, g', \ell')$ is true; at a call the edge $(g, \ell) \xrightarrow{U^{+}(f)} (g', \ell')$ indicates that $U^{+}(t, \ell, \ell', f)$ is true, and at a return the edge $(g, \ell) \xrightarrow{U^{-}(f)} (g', \ell')$ indicates that $U^{-}(t, \ell, f, \ell')$ is true; edges labeled with Sum, Sum^{+} and Sum^{-} are interpreted similarly. We explain how to infer some edges from other edges in the figures; the inferred edges are dashed.



where
– $P(t,g_2,\ell_3,g_2,\ell_3)$ *is inferred by* (CFL PUSH)
– $P(t,g_2,\ell_3,g_3,\ell_4)$ *is inferred by* (CFL STEP)
– $Sum^-(t,g_2,\ell_3,f,g_3,\ell'_2)$ *is inferred by* (CFL SUM$^-$)
– $P(t,g_1,\ell_1,g_3,\ell'_2)$ *is inferred by* (CFL POP)

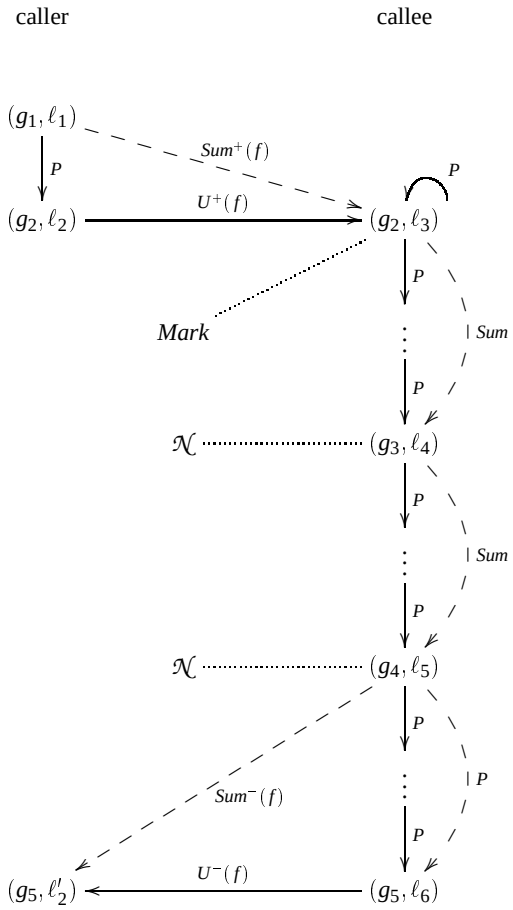
Figure 6. Application of (CFL-POP)

In Figure 6 a caller is being summarized from state (g_1,ℓ_1) to the point of call at state (g_2,ℓ_2) by the path edge $P(t,g_1,\ell_1,g_2,\ell_2)$. At the call, indicated by the edge $U^+(t,\ell_2,\ell_3,f)$, a self-loop $P(t,g_2,\ell_3,g_2,\ell_3)$ on the entry state (g_2,ℓ_3) of the callee is inferred to start off a new procedure summary. After some computation steps in the callee a return point (g_3,ℓ_4) is reached, and the summary edge $Sum^-(t,g_2,\ell_3,g_3,\ell'_2)$ is inferred, which connects the entry state of the callee to the state immediately following the return in the caller. Finally, the path edge $P(t,g_1,\ell_1,g_3,\ell'_2)$ is inferred to connect the original state (g_1,ℓ_1) to the state following the return.

It is important to note that the rules are designed to handle the complications that arise from a transaction terminating inside a function. Due to such a transaction, a summary edge may end before the return point or begin after the entry point of a callee. The rules ensure that, if a summary for a function is only partial (i.e., it does not span across both call and return), then the reachability level will execute both the call and return actions, via the PUSH and POP rules. These situations could involve scheduling other threads before, during, or after the call, as defined by the reachability relation Ω.

Figure 7 illustrates inference of a partial summary for a function in which a transaction begins at the entry state (at (g_2,ℓ_3)) but ends (at (g_3,ℓ_4)) before a return point. The end of the transaction is indicated by the fact that $\mathcal{N}(t,g_3,\ell_4)$ is true. A partial summary up to the transaction boundary is cached in the edge $Sum(t,g_2,\ell_3,g_3,\ell_4)$, which is inferred by rule (CFL SUM). Because this summary does not span across the entire call, the reachability algorithm must execute the call. This is ensured by the inference of $Mark(g_2,\ell_3)$ at the same time that the fact $Sum(t,g_2,\ell_3,g_3,\ell_4)$ is inferred by (CFL SUM). The fact $Mark(g_2,\ell_3)$ allows the inference of $Sum^+(t,g_1,\ell_1,g_2,\ell_3,f)$, which in turn is used by the reachability rule (PUSH) to execute the call. After executing the call, the partial summary edge $Sum(t,g_2,\ell_3,g_3,\ell_4)$ is available to the reachability level, via rule (STEP).

At state (g_3,ℓ_4) a new transaction summary begins via (CFL



where
– $P(t,g_2,\ell_3,g_2,\ell_3)$ *is inferred by* (CFL PUSH)
– $Sum(t,g_2,\ell_3,g_3,\ell_4)$ *is inferred by* (CFL SUM)
– $Mark(t,g_2,\ell_3)$ *is inferred by* (CFL SUM)
– $Sum^+(t,g_1,\ell_1,g_2,\ell_3,f)$ *is inferred by* (CFL SUM$^+$)
– $Sum(t,g_3,\ell_4,g_4,\ell_5)$ *is inferred by* (CFL SUM)
– $Sum^-(t,g_4,\ell_5,f,g_5,\ell'_2)$ *is inferred by* (CFL SUM$^-$)

Figure 7. Partial procedure summaries

STEP) (there is a self-loop $P(t,g_3,\ell_4,g_3,\ell_4)$ which is not shown for simplicity). The summary continues until the new transaction ends at (g_4,ℓ_5). At that point, the summary edge $Sum(t,g_3,\ell_4,g_4,\ell_5)$ is inferred by rule (CFL SUM). Finally, the summary $Sum^-(t,g_4,\ell_5,f,g_5,\ell'_2)$ is inferred for the transition from (g_4,ℓ_5) across the return point at (g_5,ℓ_6), via rule (CFL SUM$^-$).

To summarize (!), a summary edge (either *Sum*, Sum^+, or Sum^-) is computed by the summarization algorithm under any one of the following three conditions:

- When a transaction ends at an edge $P(t,g_1,\ell_1,g_2,\ell_2)$ (indicated by $\mathcal{N}(t,g_2,\ell_2)$), the rule (CFL SUM) is used to generate a *Sum* edge. In addition, we mark the start-state of the call using $Mark(t,g_1,\ell_1)$.
- Whenever a start state of a call is marked, the rule (CFL SUM$^+$) generates a Sum^+ edge at every corresponding call site, and also propagates the marking to the caller. This mark-

ing can result in additional Sum^+ edges being generated by iterated application of the rule (CFL SUM$^+$).

- When a procedure return is encountered, a Sum^- edge is generated by rule (CFL SUM$^-$).

5.2 Correctness

The correctness of our algorithm depends on conditions C1 and C2 from Section 4. However, since this algorithm computes the least fixpoint over a different set of equations, the condition C3 is modified to the following condition C3'. In order to state condition C3', we define the relation $P(t,g_1,\ell_1,g_2,\ell_2) \vdash Sum(t,g_1,\ell_1,g_3,\ell_3)$ to hold if and only if there exists a proof tree using Ruleset 3 at whose root is an application of (CFL STEP) with $P(t,g_1,\ell_1,g_2,\ell_2)$ among its premises and at one of whose leaves is an application of (CFL SUM) with $Sum(t,g_1,\ell_1,g_3,\ell_3)$ among its conclusions. We define $P(t,g_1,\ell_1,g_2,\ell_2) \vdash Sum^-(t,g_1,\ell_1,f,g_3,\ell_3)$ and $P(t,g_1,\ell_1,g_2,\ell_2) \vdash Sum^+(t,g_1,\ell_1,g_3,\ell_3,f)$ analogously (with applications of (CFL SUM$^-$) and (CFL SUM$^+$) at the leaves, respectively).

C3'. If $P(t,g_1,\ell_1,g_2,\ell_2)$ and $\mathcal{L}(t,g_2,\ell_2)$, then one of the following conditions must hold:

1. $P(t,g_1,\ell_1,g_2,\ell_2) \vdash Sum(t,g_1,\ell_1,g_3,\ell_3)$ for some g_3,ℓ_3.
2. $P(t,g_1,\ell_1,g_2,\ell_2) \vdash Sum^-(t,g_1,\ell_1,f,g_3,\ell_3)$ for some g_3,ℓ_3,f.
3. $P(t,g_1,\ell_1,g_2,\ell_2) \vdash Sum^+(t,g_1,\ell_1,g_3,\ell_3,f)$ for some g_3,ℓ_3,f.

THEOREM 2. *Let $(g_0,\ell s_0,U,U^+,U^-)$ be the instrumented multithreaded program. Let Ω be the least fixpoint of the rules in Rulesets 2 and 3. Let the conditions C1, C2, and C3' be satisfied. If $(g_0,ls_0,ss_0) \rightarrow^* (g,ls,ss)$ and $ls(t) = wrong$, then there is $(g',\ell s',ss')$ and p such that $\Omega(g',\ell s',ss')$ and $\ell s'(t) = \langle wrong,p\rangle$.*

Proof (Sketch) The proof of this theorem depends on the following lemmas.

Lemma 1: If $\Sigma(g,\ell s,ss)$ and $\ell s(t) = \langle wrong,p\rangle$, then there is $(g',\ell s',ss')$ such that $\Sigma(g',\ell s',ss')$, $\ell s'(t) = \langle wrong,p\rangle$, and $(g',\ell s'(u)) \in \mathcal{N}(u)$ for all $u \in Tid$.

Lemma 2: If $\Sigma(g,\ell s,ss)$ and $(g,\ell s(u)) \in \mathcal{N}(u)$ for all $u \in Tid$, then $\Omega(g,\ell s,ss)$.

Lemma 3: If $\Sigma(g,\ell s,ss)$ and $(g,\ell s(t)) \notin \mathcal{N}(t)$, then there are g' and ℓ' such that $P(t,g',l',g,\ell s(t))$.

Lemma 4: The condition C3' implies the condition C3.

Lemma 3 is used to prove Lemma 4. Due to Lemma 4 and the preconditions of our theorem, the preconditions of Theorem 1 are satisfied. If $(g_0,ls_0,ss_0) \rightarrow^* (g,ls,ss)$ and $ls(t) = wrong$, then from Theorem 1, we get $(g',\ell s',ss')$ and p such that $\Sigma(g',\ell s',ss')$ and $\ell s'(t) = \langle wrong,p\rangle$. From Lemma 1, we get $(g'',\ell s'',ss'')$ such that $\Sigma(g'',\ell s'',ss'')$, $\ell s''(t) = \langle wrong,p\rangle$, and $(g'',\ell s''(u)) \in \mathcal{N}(u)$ for all $u \in Tid$. From Lemma 2, we get $\Omega(g'',\ell s'',ss'')$. □

5.3 Termination

In this section, we present sufficient conditions for the least fixpoint Ω of the rules in Rulesets 2 and 3 to be finite, in which case our summarization-based model checking algorithm will terminate. These conditions are satisfied by a variety of realistic programs that use shared variables, synchronization, and recursion.

```
int g = 0, x = 0, y = 0;
mutex m;

 void foo (int r) {       void main() {
 L0:   if (r == 0) {            int q = choose({0,1});
 L1:       x = 1;       M0:   foo(q);
 L2:       y = 1;       M1:   acquire(m)
 L3:       foo(r);      M2:   assert(g >= 1);
       } else {         M3:   release(m);
 L4:     acquire(m);    M4:   return;
 L5:     g++;           }
 L6:     release(m);
       }
 L7:   return;
 }

 P = { main() } || { main() }
```

Figure 8. Nonterminating example

In our notation, a frame $f \in Frame$ corresponds to a procedure call and essentially encodes the values to which the local variables (*e.g.*, the program counter) should be set, once the procedure call returns. A frame f is *recursive* if and only if there is a transition sequence $(g_0,\ell s_0,ss_0) \rightarrow^* (g,\ell s,ss)$ and a thread t such that f occurs more than once in the stack $ss(t)$. A frame f is *transactional* if and only if for all transition sequences $(g_0,\ell s_0,ss_0) \rightarrow^* (g,\ell s,ss)$ and for all $t \in Tid$, if f occurs on the stack $ss(t)$, then $(g,\ell s(t)) \notin \mathcal{N}(t)$. If f is transactional, then in any execution, after a thread t pushes f on the stack, execution continues with all states outside $\mathcal{N}(t)$ until f is popped.

THEOREM 3. *Suppose the domains Tid, Global, Local, and Frame are all finite. If every recursive frame $f \in Frame$ is transactional, then the set of reachable states Ω is finite, and the model checking algorithm based on Rulesets 2 and 3 terminates.*

Proof (Sketch) Because the sets *Global*, *Local*, *Tid* and *Frame* are finite, it immediately follows that the relations *P*, *Sum*, Sum^+, Sum^-, and *Mark* computed by the summarization rules in Ruleset 3 are finite. Consider first the summarization rules acting on a sequence of transitions following the push of a recursive frame f. The rule (CFL SUM) cannot be applied to any premise of the form $P(t,g,\ell,g',\ell')$. The reason is that f is transactional, and therefore $(g',\ell') \notin \mathcal{N}(t)$. Hence, no facts of the form $Mark(t,g,\ell)$ or $Sum(t,g,\ell,g',\ell')$ can be deduced due to any pair (g',ℓ'). Because no such fact $Mark(t,g,\ell)$ can be deduced, the rule (CFL SUM$^+$), in turn, cannot be applied to any premise of the form $P(t,g,\ell,g',\ell')$, and hence no fact of the form $Sum^+(t,g,\ell,g',\ell',f)$ can be deduced.

Consider next the reachability rules for the set Ω acting on a sequence of states following the push of a recursive frame f. By the argument above, no facts of the form $Sum(t,g,\ell,g',\ell')$ and no facts of the form $Sum^+(t,g,\ell,g',\ell',f)$ can be deduced. It follows that only the reachability rules (INIT), (STEP), and (POP) can be applied. Because none of these rules push frames on the stacks, only finitely many facts of the form $\Omega(g,\ell s,ss)$ can be deduced from the push of a recursive frame. Since only the set *Stacks* can be infinite on a program and non-recursive frames can generate only finitely many distinct stacks, it follows that only finitely many facts $\Omega(g,\ell s,ss)$ can be deduced from a program all of whose recursive frames are transactional. □

Figure 8 shows an example program on which our two-level algorithm does not terminate. Here, the procedure `foo` is both recursive and not transactional due to the accesses it makes to the global variables x and y. As a result, Sum^+ edges are returned by the summarization algorithm for the recursive call inside procedure `foo`, and consequently the reachability algorithm does not terminate.

5.4 Single-threaded programs

In a single threaded program, we can make the set $\mathcal{N}(1)$ for the single thread contain just the initial state of the program, and the states in which the thread has gone wrong. Suppose the program does not reach an error state. Then, the rule (CFL SUM) can never be applied and the summarization rules will never generate *Sum* or Sum^+ edges. Consequently, the reachability rules will never explore states in which the stack is non-empty, and the model checking algorithm with summarization specializes to CFL reachability. The summary of a procedure contains only Sum^- edges and is identical to the summary produced by the CFL reachability algorithm.

6 Related work

Recently, several papers have used the idea of reduction to develop analyses for concurrent programs. Flanagan and Qadeer developed a type system leveraging the ideas of reduction and transactions to verify atomicity in multithreaded programs [15, 14]. Their type system was inspired by the Calvin-R static checking tool [16]. Calvin-R supports modular verification of multithreaded programs by annotating each procedure with a specification; this specification is related to the procedure implementation via an abstraction relation that combines the notions of simulation and reduction. Recently, Flanagan and Freund have also developed a dynamic atomicity checker called Atomizer for multithreaded Java programs [11].

A number of dataflow techniques have been devised to analyze programs with both concurrency and procedure calls. Duesterwald and Soffa use a system of dataflow equations to check if two statements in a concurrent program can potentially execute in parallel [7]. 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 [8]. 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 [20].

Ramalingam proved the undecidability of assertion checking with both concurrency and procedure calls [18]. The proof is by reduction from the undecidable problem of checking the emptiness of the intersection of two context-free languages. 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. In contrast, our work operates on the concrete multithreaded program and uses summaries to gain reuse, scalability, and termination in a number of cases.

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.

For restricted models of synchronization, such as fork-join synchronization, assertion checking is decidable even with both concurrency and procedure calls. Esparza and Podelski present an algorithm for this restricted class of programs [9].

Counter machines and variants of Petri nets have been used to check assertions on concurrent programs with unbounded number of threads [6, 2]. However, these methods handle procedure calls by inlining.

7 Conclusions

We have presented a novel model checking algorithm to check assertions on multithreaded programs with procedure calls. Inspired by procedure summarization in sequential programs, our algorithm attempts to use summaries to obtain reuse and scalability. Our algorithm functions in two levels. The first level performs reachability analysis and maintains an explicit stack for each thread. The second level computes a summary for each procedure. Under certain conditions (stated precisely in Theorem 3), we guarantee that our two-level algorithm will terminate even in the presence of recursion and concurrency. We are currently implementing our algorithm in a new model checker called ZING being developed at Microsoft Research.

8 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. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031, pages 158–173. Springer-Verlag, 2001.

[3] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.

[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] 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.

[6] G. Delzanno, J-F. Raskin, and L. V. Begin. Towards the automated verification of multithreaded Java programs. In *TACAS 02: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2280, pages 173–187. Springer-Verlag, 2002.

[7] 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.

[8] 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.

[9] 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.

[10] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *CAV 01: Computer Aided Verficiation*, LNCS 2102, pages 324–336. Springer-Verlag, 2001.

[11] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL 04: Principles of Programming Languages*. ACM, 2004.

[12] C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN 03: SPIN Workshop*, LNCS 2648, pages 213–225. Springer-Verlag, 2003.

[13] C. Flanagan and S. Qadeer. Transactions for software model checking. In *SoftMC 03: Software Model Checking Workshop*, 2003.

[14] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI 03: Programming Language Design and Implementation*, pages 338–349. ACM, 2003.

[15] C. Flanagan and S. Qadeer. Types for atomicity. In *TLDI 03: Types in Language Design and Implementation*, pages 1–12. ACM, 2003.

[16] S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. In *FTfJP 03: Formal Techniques for Java-like Programs*, 2003.

[17] R. J. Lipton. Reduction: A method of proving properties of parallel programs. In *Communications of the ACM*, volume 18:12, pages 717–721, 1975.

[18] G. Ramalingam. Context sensitive synchronization sensitive analysis is undecidable. *ACM Trans. on Programming Languages and Systems*, 22:416–430, 2000.

[19] 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.

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

[21] 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.

[22] B. Steffen and O. Burkart. Composition, decomposition and model checking optimal of pushdown processes. *Nordic Journal of Computing*, 2(2):89–125, 1995.