Sequentializing Parameterized Programs [*]

Salvatore La Torre
Dipartimento di Informatica
Università degli Studi di Salerno, Italy
slatorre@unisa.it

P. Madhusudan
Department of Computer Science
University of Illinois at Urbana-Champaign, USA
madhu@cs.illinois.edu

Gennaro Parlato
Department of Electronic & Computer Systems
University of Southampton, UK
gennaro@ecs.soton.ac.uk

We exhibit assertion-preserving (reachability preserving) transformations from parameterized concurrent shared-memory programs, under a k-round scheduling of processes, to sequential programs. The salient feature of the sequential program is that it tracks the local variables of only one thread at any point, and uses only $O(k)$ copies of shared variables (it does not use extra counters, not even one counter to keep track of the number of threads). Sequentialization is achieved using the concept of a linear interface that captures the effect *an unbounded block* of processes have on the shared state in a k-round schedule. Our transformation utilizes linear interfaces to sequentialize the program, and to ensure the sequential program explores only reachable states and preserves local invariants.

1 Introduction

The theme of this paper is to build verification techniques for parameterized concurrent shared-memory programs: programs with *unboundedly* many threads, many of them running identical code that concurrently evolve and interact through shared variables. Parameterized concurrent programs are extremely hard to check for errors. Concurrent shared-memory programs with a finite number of threads are already hard, and extending existing methods for sequential programs, like Floyd-Hoare style deductive verification, abstract interpretation, and model-checking, is challenging. Parameterized concurrent programs are even harder.

A recent proposal in the verification community to handle concurrent program verification is to *reduce the problem to sequential program verification*. Of course, this is not possible, in general, unless the sequential program tracks the entire configuration of the concurrent program, including the local state of each thread. However, recent research has shown efficient sequentializations for concurrent programs with finitely many threads when restricted to a *fixed number of rounds of schedule* (or a fixed number of context-switches) [15, 12]. A round of schedule involves scheduling each process, one at a time in some order, where each process is allowed to take an *arbitrary* number of steps.

The appeal of sequentialization is that by reducing concurrent program verification to sequential program verification, we can bring to bear all the techniques and tools available for the analysis of sequential programs. Such sequentializations have been used recently to convert concurrent programs under bounded round scheduling to sequential programs, followed by automatic deductive verification techniques based on SMT solvers [14], in order to find bugs (see also [6]). The goal of this paper is to find a similar translation for *parameterized* concurrent programs where the number of threads is unbounded.

[*] This research was supported in part by MIUR-FARB 2009-2011 Università degli Studi di Salerno and NSF Career Award 0747041.

S. Bauer, J.-B. Raclet (Eds.): 4th International Workshop
on Foundations of Interface Technologies (FIT 2012)
EPTCS 87, 2012, pp. 34–47, doi:10.4204/EPTCS.87.4

Again, we are looking for an efficient translation— to convert a parameterized concurrent program to a sequential program so that the latter tracks *the local state of at most one thread at any time, and uses only a bounded number of copies of shared variables*.

The motivation for the bounded round restriction is inspired from recent research in testing that suggests that most concurrency errors manifest themselves within a few context-switches (executions can be, however, arbitrarily long). The CHESS tool from Microsoft Research, for example, tests concurrent programs only under schedules that have a bounded number of context-switches (or pre-emptions) [17]. In the setting where there is an unbounded number of threads, the natural extension of bounded context-switching is bounded-round context-switching, as the latter executes schedules that allow all threads in the system to execute (each thread is context-switched into a bounded number of times). Checking a parameterized program to be correct for a few rounds gives us considerable confidence in its correctness.

The main result of this paper is that efficient sequentializations of *parameterized* programs are also feasible, when restricted to bounded-round context-switching schedules. More precisely, we show that given a parameterized program P with an unbounded number of threads executing under k-round schedules and an error state e, there is an effectively constructible (non-deterministic) sequential program S with a corresponding error state e' that satisfies the following: (a) the error state e is reachable in P iff the error state e' is reachable in S, (b) a localized state (the valuation of a thread's local variables and the shared variables) is reachable in P iff it is reachable in S— in other words, the transformation preserves assertion invariants and explores only reachable states, (we call such a transformation *lazy*), and (c) S at any point tracks the local state of only one thread and at most $O(k)$ copies of shared variables, and furthermore, uses no additional unbounded memory such as counters or queues.

The existence of a transformation of the above kind is theoretically challenging and interesting. First, when simulating a parameterized program, one would expect that *counters* are necessary— for instance, in order to simulate the second round of schedule, it is natural to expect that the sequential program must remember at least the number of threads it used in the first round. However, our transformation does not introduce counters. Second, a lazy transformation is hard as, intuitively, the sequential program needs to *return* to a thread in order to simulate it, and yet cannot keep the local state during this process. The work reported in [12] achieves such a lazy sequentialization for concurrent programs with finitely many threads using *recomputation* of local states. Intuitively, the idea is to fix a particular ordering of threads, and to restart threads which are being context-switched into to recompute their local state. Sequentializing parameterized concurrent programs is significantly harder as we, in addition, do not even know how many threads were active or how they were ordered in an earlier round, let alone know the local states of these threads.

Our main technical insight to sequentialization is to exploit a concept called *linear interface* (introduced by us recently [13] to provide model-checking algorithms for *Boolean* parameterized systems). A linear interface summarizes the effect of an *unbounded block of processes* on the shared variables in a k-rounds schedule. A linear interface is of the form $(\overline{u},\overline{v})$ where $\overline{u}$ and $\overline{v}$ are k-tuples of valuations of shared variables. A block of threads have a linear interface $(\overline{u},\overline{v})$ if, intuitively, there is an execution which allows context-switching into this block with u_i and context-switch out at v_i, for i growing consecutively from 1 to k, *while preserving the local state* across the context-switches (see Figure 1).

In classic verification of sequential programs with recursion (in both deductive verification as well as model-checking), the idea of summarizing procedures using pre-post conditions (or summaries in model-checking algorithms) is crucial in handling the infinite recursion depth. In parameterized programs, linear interfaces have a similar role but across a concurrent dimension: they capture the pre-post condition for *a block of unboundedly many processes*. However, because a block of processes has a persistent state across two successive rounds of a schedule (unlike a procedure called in a sequential program), we cannot summarize the effect of a block using a pre-post predicate that captures a set of pairs of the form (u,v). A linear interface captures a sequence of length k of pre-post conditions, thus capturing in essence the effect of the local states of the block of processes that is adequate for a k-round schedule.

Our sequentialization synthesizes a sequential program that uses a recursive procedure to compute linear interfaces. Intuitively, linear interfaces of the parameterized program correspond to *procedural summaries* in the sequential program. Our construction hence produces a recursive program even when the parameterized program has no recursion. However, the translation also works when the parameterized program has recursion, as the program's recursion gets properly nested within the recursion introduced by the translation.

Our translations work for general programs, with unbounded data-domains. When applied to parameterized programs over *finite data-domains*, it shows that a class of *parameterized pushdown systems* (finite automata with an unbounded number of stacks) working under a bounded round schedule, can be simulated faithfully by a *single-stack pushdown system* (see Section 4.2).

The sequentialization provided in this paper paves the way to finding errors in parameterized concurrent programs with unbounded data domains, up to a bounded number of rounds of schedule, as it allows us to convert them to sequential programs, in order to apply the large class of sequential analysis tools available—model-checking, testing, verification-condition generation based on SMT solvers, and abstraction-based verification. The recent success in finding errors in concurrent systems code (for a finite number of threads) using a sequentialization and then SMT-solvers [14, 6] lends credence to this optimism.

Related work. The idea behind bounding context-switches is that most concurrency errors manifest within a few switches [20, 16]. The CHESS tool from Microsoft espouses this philosophy by testing concurrent programs by systematically choosing all schedules with a small number of preemptions [17]. Theoretically, context-bounded analysis was motivated for the study of concurrent programs with bounded data-domains and recursion, as it yielded a decidable reachability problem [19], and has been exploited in model-checking [21, 15, 11]. In recent work [13], we have designed model-checking algorithms for Boolean abstractions of parameterized programs using the concept of linear interfaces; these work only for bounded data domains and also do not give sequentializations.

The first sequentialization of concurrent programs was proposed for a finite number of threads and two context-switches [20], followed by a general *eager* conversion that worked for arbitrary number of context-switches [15], and a *lazy* conversion proposed by us in [12]. Sequentialization has been used recently on concurrent device drivers written in C with dynamic heaps, followed by using proof-based verification techniques to find bugs [14]. A sequentialization for delay-bounded schedulers that allows exploration of concurrent programs with dynamic thread creation has been discovered recently [6]. Also, in recent work, it has been established that any concurrent program with finitely many threads that can be reasoned with compositionally, using a rely-guarantee proof, can be sequentialized [7].

A recent paper proposes a solution using Petri net reachability to the reachability problem in concurrent programs with *bounded data domains* and dynamic creation of threads, where a thread is context-switched into only a bounded number of times [1]. Since dynamic thread creation can model unboundedly many threads, this framework is more powerful (and much more expensive in complexity) than ours, *when restricted to bounded data-domains*.

There is a rich history of verifying parameterized asynchronously communicating concurrent programs, especially motivated by the verification of distributed protocols. We do not survey these in detail (see [4, 10, 9, 5, 18, 2], for a sample of this research).

2 Preliminaries

Sequential recursive programs. Let us fix the syntax of a simple *sequential* programming language with variables ranging over only the integer and Boolean domains, with explicit syntax for nondeterminism, and (recursive) function calls. For simplicity of exposition, we do not consider dynamically allocated structures or domains other than integers; however, all results in this paper can be easily extended to handle such features.

Sequential programs are described by the following grammar:

$$
\begin{array}{lll}
\langle \textit{seq-pgm} \rangle & ::= & \langle \textit{vardec} \rangle ; \langle \textit{sprocedure} \rangle^{*} \\
\langle \textit{vardec} \rangle & ::= & \langle \textit{type} \rangle\ x \mid \langle \textit{vardec} \rangle ; \langle \textit{vardec} \rangle \\
\langle \textit{type} \rangle & ::= & \texttt{int} \mid \texttt{bool} \\
\langle \textit{sprocedure} \rangle & ::= & (\langle \textit{type} \rangle \mid \texttt{void})\, f(x_1,\ldots,x_h)\ \texttt{begin}\ \langle \textit{vardec} \rangle ; \langle \textit{seq-stmt} \rangle\ \texttt{end} \\
\langle \textit{seq-stmt} \rangle & ::= & \langle \textit{seq-stmt} \rangle ; \langle \textit{seq-stmt} \rangle \mid \texttt{skip} \mid x := \langle \textit{expr} \rangle \mid \texttt{assume}(\langle \textit{b-expr} \rangle) \mid \\
 & & \texttt{call}\ f(x_1,\ldots,x_h) \mid \texttt{return}\ x \mid \texttt{while}\ (\langle \textit{b-expr} \rangle)\ \texttt{do}\ \langle \textit{seq-stmt} \rangle\ \texttt{od} \\
 & & \texttt{if}\ (\langle \textit{b-expr} \rangle)\ \texttt{then}\ \langle \textit{seq-stmt} \rangle\ \texttt{else}\ \langle \textit{seq-stmt} \rangle\ \texttt{fi} \mid \texttt{assert} \langle \textit{b-expr} \rangle \\
\langle \textit{expr} \rangle & ::= & x \mid c \mid f(y_1,\ldots,y_h) \mid \langle \textit{b-expr} \rangle \\
\langle \textit{b-expr} \rangle & ::= & T \mid F \mid * \mid x \mid \neg \langle \textit{b-expr} \rangle \mid \langle \textit{b-expr} \rangle \vee \langle \textit{b-expr} \rangle
\end{array}
$$

Variables are scoped in two ways, either as global variables shared between procedures, or variables local to a procedure, according to where they are declared. Functions are all call-by-value. Some functions f may be interpreted to have existing functionality, such as integer addition or library functions, in which case their code is not given and we assume they happen atomically. We assume the program is well-typed according to the type declarations.

Note that Boolean expressions can be true, false, or non-deterministically true or false ($*$), and hence programs are non-deterministic (which will be crucial as we will need to simulate concurrent programs, which can be non-deterministic). These non-deterministic choices can be replaced as *inputs* in a real programming language if we need to verify the sequential program.

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

The `assert` statements form the specification for the program, and express invariants that can involve all variables in scope. Note that *reachability* of a particular statement can be encoded using an `assert F` at that statement.

Parameterized programs with a fixed number of shared variables. We are interested in concurrent programs composed of several concurrent processes, each executing on possibly unboundedly many threads (*parameterized programs*). All threads run in parallel and share a fixed number of variables.

A *concurrent process* is essentially a sequential program with the possibility of declaring sets of statements to be executed *atomically*, and is given by the following grammar (defined as an extension on the syntax for sequential programs):

$$
\begin{array}{lll}
\langle \textit{process} \rangle & ::= & \texttt{process}\ P\ \texttt{begin}\ \langle \textit{vardec} \rangle ; \langle \textit{cprocedure} \rangle^{*}\ \texttt{end} \\
\langle \textit{cprocedure} \rangle & ::= & (\langle \textit{type} \rangle \mid \texttt{void})\, f(x_1,\ldots,x_h)\ \texttt{begin}\ \langle \textit{vardec} \rangle ; \langle \textit{conc-stmt} \rangle\ \texttt{end} \\
\langle \textit{conc-stmt} \rangle & ::= & \langle \textit{conc-stmt} \rangle ; \langle \textit{conc-stmt} \rangle \mid \langle \textit{seq-stmt} \rangle \mid \texttt{atomic begin}\ \langle \textit{seq-stmt} \rangle\ \texttt{end}
\end{array}
$$

The syntax for parameterized programs is obtained by adding the following rules:

$\langle \textit{param-pgm} \rangle$ ::= $\langle \textit{vardec} \rangle \langle \textit{init} \rangle \langle \textit{process} \rangle^*$
$\langle \textit{init} \rangle$ ::= $\langle \textit{seq-stmt} \rangle$

Variables in a parameterized program can be scoped locally, globally (i.e. to a process at a particular thread) or shared (shared amongst all processes in all threads, when declared before `init`). The statements and assertions in a parameterized program can refer to all variables in scope.

Each parameterized program has a sequential block of statements, `init`, where the shared variables are initialized. The parameterized program is initialized with an arbitrary finite number of threads, each thread running a copy of one of the processes. Dynamic creation of threads is not allowed. However, dynamic creation can be modeled (at the cost of a context-switch per created thread) by having threads created in a “dormant” state, which get active when they get a message from the parent thread to get created.

An *execution* of a parameterized program is obtained by interleaving the behaviors of the threads which are involved in it.

Formally, let $\mathscr{P} = (S, \texttt{init}, \{P_i\}_{i=1}^n)$ be a *parameterized program* where S is the set of shared variables and P_i is a process for $i = 1, \ldots, n$. We assume that each statement of the program has a unique *program counter* labeling it. A *thread* T of $\mathscr{P}$ is a copy (instance) of some P_i for $i = 1, \ldots, n$. At any point, only one thread is *active*. For any $m > 0$, a *state* of $\mathscr{P}$ is denoted by a tuple $(map, i, s, \sigma_1, \ldots, \sigma_m)$ where: (1) $map : [1,m] \rightarrow P$ is a mapping from threads $T_1, \ldots T_m$ to processes, (2) the thread which is currently active is T_i, where $i \in [1,m]$ (3) s is a valuation of the shared variables, and (4) σ_j (for each $j \in [1,m]$) is a local state of T_j. Note that each σ_j is a *local state* of a process, and is composed of a valuation of the program counter, local, and global variables of the process, and a *call-stack* of local variable valuations and program counters to model procedure calls.

At any state $(map, i, s, \sigma_1, \ldots, \sigma_m)$, the valuation of the shared variables s is referred to as the *shared state*. A *localized state* is the *view* of the state by the current process, i.e. it is $(\widehat{\sigma}_i, s)$, where $\widehat{\sigma}_i$ is the component of σ_i that defines the valuation of local and global variables, and the local pc (but not the call-stack), and s is the valuation of the shared variables in scope. Note that assertions express properties of the localized state only. Also, note that when a thread is not scheduled, the local state of its process does not change.

The interleaved semantics of parameterized programs is given in the obvious way. We start with an arbitrary state, and execute the statements of `init` to prepare the initial shared state of the program, after which the threads become active. Given a state $(map, i, \nu, \sigma_1, \ldots, \sigma_m)$, it can either fire a *transition* of the process at thread T_i (i.e., of process $map(i)$), updating its local state and shared variables, or *context-switch* to a different active thread by changing i to a different thread-index, provided that in T_i we are not in a block of sequential statements to be executed atomically.

Verification under bounded round schedules: Fix a parameterized program $\mathscr{P} = (S, \texttt{init}, \{P_i\}_{i=1}^n)$. The verification problem asks, given a parameterized program $\mathscr{P}$, whether every execution of the program respects all assertions.

In this paper, we consider a restricted verification problem. A k-round schedule is a schedule that, for some ordering of the threads $T_1, \ldots, T_m$, activates threads in k rounds, where in each round, each thread is scheduled (for any number of steps) according to this order. Note that an execution under a k-round schedule (k*-round execution*) can execute an unbounded number of steps. Given a parameterized program and $k \in \mathbb{N}$, the verification problem for parameterized programs under bounded round schedules asks whether any assertion is violated in some k-round execution.

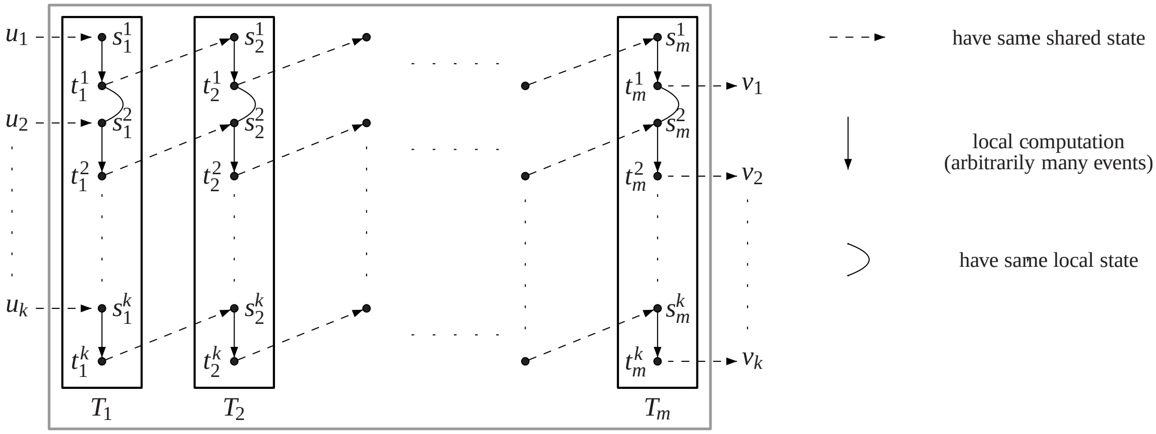

Figure 1: A linear interface

3 Linear interfaces

We now introduce the concept of a linear interface, which captures the effect a block of processes has on the shared state, when involved in a k-round execution. The notion of linear interfaces will play a major role in the lazy conversion to sequential programs.

We fix a parameterized program $\mathscr{P} = (S, \texttt{init}, \{P_i\}_{i=1}^n)$ and a bound $k > 0$ on the number of rounds.

Notation: let $\overline{u} = (u_1, \ldots, u_k)$, where each u_i is a shared state of $\mathscr{P}$.

A pair of k-tuples of shared variables $(\overline{u}, \overline{v})$ is a *linear interface* of length k (see Figure 1) if: **(a)** there is an ordered block of threads $T_1, \ldots, T_m$ (running processes of $\mathscr{P}$), **(b)** there are k rounds of execution, where each execution starts from shared state u_i, exercises the threads in the block one by one, and ends with shared state v_i (for example, in Figure 1, the first round takes u_1 through states s_1^1, t_1^1, s_2^1, $t_2^1, \ldots$ to t_m^1 where the shared state is v_1), and **(c)** the local state of threads is preserved between consecutive rounds (in Figure 1, for example, t_1^1 and s_1^2 have the same local state). Informally, a linear interface is the *effect* a block of threads can have on the shared state in a k-round execution, in that they transform $\overline{u}$ to $\overline{v}$ across the block.

Formally, we have the following definition (illustrated by Figure 1).

Definition 1 *(*LINEAR INTERFACE*) [13]*
Let $\overline{u} = (u_1, \ldots, u_k)$ and $\overline{v} = (v_1, \ldots, v_k)$ be tuples of k shared states of a parameterized program $\mathscr{P}$ (with processes P).
The pair $(\overline{u}, \overline{v})$ is a linear interface *of $\mathscr{P}$ of length k if there is some number of threads $m \in \mathbb{N}$, an assignment of threads to processes $map : [1,m] \rightarrow P$ and states $s_i^j = (map, i, x_i^j, \sigma_1^{i,j}, \ldots, \sigma_m^{i,j})$ and $t_i^j = (map, i, y_i^j, \gamma_1^{i,j}, \ldots, \gamma_m^{i,j})$ of $\mathscr{P}$ for $i \in [1,m]$ and $j \in [1,k]$, such that for each $i \in [1,m]$:*

- *$x_1^j = u_j$ and $y_m^j = v_j$, for each $j \in [1,k]$;*
- *t_i^j is reachable from s_i^j using only local transitions of process $map(i)$, for each $j \in [1,k]$;*
- *$\sigma_i^{i,1}$ is an initial local state for process $map(i)$;*
- *$\sigma_i^{i,j+1} = \gamma_i^{i,j}$ for each $j \in [1,k-1]$ (local states are preserved across rounds);*
- *$x_{i+1}^j = y_i^j$, except when $i = m$ (shared states are preserved between context-switches of a single round);*
- *(t_i^j, s_{i+1}^j), except when $i = m$, is a context-switch.* □

Note that the above definition of a linear interface places no restriction on the relation between v_j and u_{j+1}— all that we require is that the block of threads must take as input $\overline{u}$ and compute $\overline{v}$ in the k rounds, preserving the local configuration of threads between rounds.

A linear interface $(\overline{u},\overline{v})$ of length k is *wrapped* if $v_i = u_{i+1}$ for each $i \in [1,k-1]$, and is *initial* if $u_1$is an initial shared state of $\mathscr{P}$.

For a wrapped initial linear interface, from the definition of linear interfaces it follows that the k pieces of execution demanded in the definition can be stitched together to get a complete execution of the parameterized program, that starts from an initial state. We say that an execution *conforms* to a particular linear interface if it meets the condition demanded in the definition.

Lemma 1 *[13] Let $\mathscr{P}$ be a parameterized program. An execution of $\mathscr{P}$ is under a k-round schedule iff it conforms to some wrapped initial linear interface of $\mathscr{P}$ of length k.* □

Hence to verify a program $\mathscr{P}$ under k-round schedules, it suffices to check for failure of assertions along executions that conform to some wrapped initial interface of length k.

4 Sequentializing parameterized programs

In this section, we present a sequentialization of parameterized programs that preserves assertion satisfaction. Our translation is “lazy” in that the states reachable in the resulting program correspond to reachable states of the parameterized program. Thus, it preserves invariants across the translation: an invariant that holds at a particular statement in the concurrent program will hold at the corresponding statement in the sequential program.

A simpler *eager* sequentialization scheme for parameterized programs that reduces reachability of error states for parameterized programs but explores *unreachable states* as well, can be obtained by a simple adaptation of the translation from concurrent programs with finitely many threads to sequential programs given in [15]. This scheme consists of simulating each thread till completion across *all the rounds*, before switching to the next thread, and then, at the end, checking if the execution of all the threads corresponds to an actual execution of the parameterized program. Nondeterminism is used to guess the number of threads, the schedule, and the shared state at the beginning of each round. However, this translation explores unreachable states, and hence does not preserve assertions across the translation.

Motivating laziness: A lazy translation that explores only localized states reachable by a parameterized program has obvious advantages over an eager translation. For example, if we subject the sequential program to model-checking using state-space exploration, the lazy sequentialization has fewer reachable states to explore. The lazy sequentialization has another interesting consequence, namely that the sequential program will not explore unreachable parts of the state-space where invariants of the parameterized program get violated or where executing statements leads to system errors due to undefined semantics (like division-by-zero, buffer-overflows, etc.), as illustrated by the following example.

Example 1 *Consider an execution of the parameterized program $\mathscr{P}$ from Figure 2. The program involves only two threads: T_1 which executes P_1 and T_2 which executes P_2. Observe that any execution of T_1 cycles on the while-loop until T_2 sets* blocked *to false. But before this, T_2 sets y to* 2 *and hence the assertion $(y \neq 0)$ is true in P_1. However, in an execution of the simpler eager sequentialization, we would simulate P_1 for k rounds and then simulate P_2 through k rounds. In order to simulate P_1, the eager translation would* guess *non-deterministically a k-tuple of shared variables $u_1,\ldots,u_k$. Consider an execution where u_1 assigns* blocked *to be true, and u_2 assigns* blocked *to false and y to* 0*. The sequential program*

```
bool blocked := T ;
int x := 0, y := 0;

process P1:                      process P2:
  main() begin                     main() begin
    while (blocked) do               x := 12;
       skip;                         y := 2;
    od                               blocked := F ;  //unblock P1
    assert(y!=0);                  end
    x := x/y;
  end
```

Figure 2: Assertion not preserved by the eager sequentialization.

would, in its simulation of P_1 in the first round, reach the while-loop, and would jump to the second round to simulate P_1 from u_2. Note that the assertion condition would fail, and will be duly noted by the sequential program. But if the assertion wasn't there, the sequentialization would execute the statement $x := x/y$, which would results in a "division by zero" exception. In short, $(y \neq 0)$ is not an invariant for the statement $x := x/y$ in the eager sequentialization. The lazy translation presented in the next section avoids such scenarios. □

4.1 Lazy sequentialization

Without loss of generality, we fix a parameterized program $\mathscr{P} = (S, \texttt{init}, \{P\})$ over one process. Note that this is not a restriction, as we can always build P so that it makes a non-deterministic choice at the beginning, and decides to behave as one of a set of processes. We also replace functions with return values to `void` functions that communicate the return value to the caller using global (unshared) variables. Finally, we fix a bound $k > 0$ on the number of rounds.

We perform a lazy sequentialization of a parameterized program $\mathscr{P}$ by building a sequential program that computes linear interfaces. More precisely, at the core of our construction is a function `linear_int` that takes as input a set of valuations of shared variables $\langle u_1, \ldots, u_i, v_1, \ldots v_{i-1} \rangle$ (for some $1 \leq i \leq k$) and *computes* a shared valuation s such that $(\langle u_1, \ldots, u_i \rangle, \langle v_1, \ldots, v_{i-1}, s \rangle)$ is a linear interface. We outline how this procedure works below.

The procedure `linear_int` will require the following pre-condition, and meet the following post-condition and invariant when called with the input $\langle u_1, \ldots, u_i, v_1, \ldots v_{i-1} \rangle$:

Precondition: There is some v_0, an initial shared state, such that $(\langle v_0, v_1, \ldots v_{i-1} \rangle, \langle u_1, u_2, \ldots u_i \rangle)$ is a linear interface.

Postcondition: The value of the shared state at the return, s, is such that $(\langle u_1, \ldots, u_i \rangle, \langle v_1, \ldots v_{i-1}, s \rangle)$ is a linear interface.

Invariant: At any point in the execution of `linear_int`, if the localized state is $(\widehat{\sigma}, s)$, and a statement of the parameterized program is executed from this state, then $(\widehat{\sigma}, s)$ is a localized state reached in some execution of $\mathscr{P}$.

Intuitively, the pre-condition says that there must be a "left" block of threads where the initial computation can start, and which has a linear interface of the above kind. This ensures that all the u_i's are indeed reachable in some computation. Our goal is to build `linear_int` to sequentially compute, using nondeterminism, any possible value of s such that $(\langle u_1,\ldots,u_i\rangle,\langle v_1,\ldots v_{i-1},s\rangle)$ is a linear interface (as captured by the post-condition). The invariant above assures laziness; recall that the laziness property says that no statement of the parameterized program will be executed on a localized state of the sequential program that is unreachable in the parameterized program.

Let us now sketch how `linear_int` works on an input $\langle u_1,\ldots,u_i,v_1,\ldots v_{i-1}\rangle$. First, it will decide non-deterministically whether the linear interface is for a *single thread* (by setting a variable *last* to T, signifying it is simulating the last thread) or whether the linear interface is for a block of threads more than one (in which case *last* is set to F).

It will start with the state (σ_1,u_1) where σ_1 is an initial local state of P, and simulate an arbitrary number of moves of P, and stop this simulation at some point, non-deterministically, ending in state (σ'_1,u'_1). At this point, we would like the computation to "jump" to state (σ'_1,u_2), however we need first to ensure that this state is reachable.

If $last = T$, i.e. if the thread we are simulating is the last thread, then this is easy, as we can simply check if $u'_1 = v_1$. If $last = F$, then `linear_int` determines whether (σ'_1,u_2) is reachable by *calling itself recursively* on the tuple $\langle u'_1\rangle$, getting the return value s, and checking whether $s = v_1$. In other words, we claim that (σ'_1,u_2) is reachable in the parameterized program if (u'_1,v_1) is a linear interface.

Here is a proof sketch. Assume (u'_1,v_1) is a linear interface; then by the pre-condition we know that there is an execution starting from a shared initial state to the shared state u_1. By switching to the current thread T_h and using the local computation of process P just witnessed, we can take the state to u'_1 (with local state σ'_1), and since (u'_1,v_1) is a linear interface, we know there is a "right" block of processes that will somehow take us from u'_1 to v_1. Again by the pre-condition, we know that we can continue the computation in the second round, and ensure that the state reaches u_2, at which point we switch to the current thread T_h, to get to the local state (σ'_1,u_2).

The above argument is the crux of the idea behind our construction. In general, when we have reached a local state (σ'_i,u'_i), `linear_int` will call itself on the tuple $u'_1,\ldots,u'_i,v_1,\ldots v_{i-1}$, get the return value s and check if $s = v_i$, before it "jumps" to the state (σ'_i,u_{i+1}). Note that when it calls itself, it maintains the pre-condition that there is a v_0 such that $(\langle v_0,v_1,\ldots v_{i-1}\rangle,\langle u'_1,\ldots,u'_i\rangle)$ is a linear interface by virtue of the fact that the pre-condition to the current call holds, and by the fact that the values $u'_1,\ldots,u'_i$ were computed consistently in the current thread.

The soundness of our construction depends on the above argument. Notice that the laziness invariant is maintained because the procedure calls itself to check if there is a "right" block whose linear interface will witness reachability, and the computation involved in this is assured to meet only reachable states because of its pre-condition which demands that there is a "left"-block that assures an execution.

Completeness of the sequentialization relies on several other properties of the construction. First, we require that a call to `linear_int` returns *all possible values* of s such that $(\langle u_1,\ldots,u_i\rangle,\langle v_1,\ldots v_{i-1},s\rangle)$ is a linear interface. Moreover, we need that for every execution corresponding to this linear interface and every local state (σ,s) seen along such an execution, the local state is met along some run of `linear_int`. It is not hard to see that the above sketch of `linear_int` does meet these requirements.

Notice that when simulating a particular thread, two successive calls to `linear_int` may result in different depths of recursive calls to `linear_int`, which means that a different number of threads are explored. However, the correctness of the computation does not depend on this, as correctness only relies on the fact that `linear_int` computes a linear interface, and the number of threads in the block that witnesses this interface is immaterial. This property of a linear interface that encapsulates a block of threads no matter how their internal composition is, is what makes a sequentialization without extra counters possible.

We will have a `main` function that drives calls to `linear_int`, calling it to compute linear interfaces

```
Denote $\overline{q}_{i,j} = q_i,\ldots,q_j$
Let $s$ be the shared variables and $g$ the global variables of $\mathscr{P}$;
bool $atom, terminate$;

main()
begin
  Let $q_1,\ldots,q_k$ be of type of $s$;
  int $i = 1$;
  $atom := F$;
  call init();
  $q_1 := g$;
  while $(i \leq k)$ do
    $terminate := F$;
    call linear_int($\overline{q}_{1,k}, \overline{q}_{2,k}, i$);
    $i$++;
    if $(i \leq k)$ then
      $q_i := s$;
    fi
  od
  return;
end
```

```
Interlined code:
if ($terminate$) then return; fi
if ($\neg atom$) then
  while (*) do
    if ($last$) then
      if ($j = bound$) then
        $terminate := T$; return;
      else assume($q'_j = s$);
        $j$++;   $s := q_j$;
      fi
    else
      $q_j := s$;  $save := g$;
      call linear_int($\overline{q}_{1,k}, \overline{q}'_{1,k-1}, j$);
      if ($j = bound$) then  return;
      else  assume($q'_j = s$);  $g := save$;
        $terminate := F$; $j$++; $s := q_j$;
      fi
    fi
  od
fi
```

Figure 3: Function main and interlined control code of the sequential program $\mathscr{P}_k^{lazy}$.

starting from a shared state u_1 that is an initial shared state. Using successive calls, it will construct linear interfaces of the form $\langle u_1,\ldots,u_i,v_1,\ldots,v_i\rangle$ maintaining that $v_j = u_{j+1}$, for each $j < i$. This will ensure that the interfaces it computes are *wrapped* interfaces, and hence the calls to linear_int meet the latter's pre-condition. When it has computed a complete linear interface of length k, it will stop, as any localized state reachable in a k-round schedule would have been seen by then (see Lemma 1).

The syntactic transformation. The sequential program $\mathscr{P}_k^{lazy}$ obtained from $\mathscr{P}$ in the lazy sequentialization consists of the function init of $\mathscr{P}$, a new function main, a function linear_int, and for every function f other than main in $\mathscr{P}$, a function f^{lazy}. The function main of $\mathscr{P}_k^{lazy}$ is shown in Figure 3. The function linear_int is obtained by transforming the main function in the process of the parameterized program, by interlining the code shown in Figure 3 between every two statements. Each functions f^{lazy} is obtained from f similarly by inserting the same interlined code. Clearly, in these transformations each call to f gets replaced with a call to f^{lazy}.

The interlined code allows to interrupt the simulation of a thread (provided we are not in an atomic section), and either jump directly to the next shared state (if $last = 1$) or call recursively linear_int to ensure that jumping to the next shared state will explore a reachable state. Observe that before calling linear_int recursively from the interlined code, we copy g (i.e., the value of P's global variables) to the local variables *save*, and after returning, copy it back to g to restore the local state.

The global variables of $\mathscr{P}_k^{lazy}$ includes all global and shared variables of $\mathscr{P}$, as well as two extra global Boolean variables *atom* and *terminate*. The variable *atom* is used to flag that the simulation is within an atomic block of instructions where context-switches are prohibited. The variable *terminate* is used to force the return from the most recent call to linear_int in the call stack (thus all the function calls which are in the call stack up to this call are also returned). This variable is set false in the beginning and after returning each call to `linear_int`.

Function `main` uses k copies of the shared variables denoted with $q_1, \dots, q_k$. It calls `init` and then iteratively calls `linear_int` with $i = 1, \dots, k$. Variable q_1 is assigned in the beginning and at each iteration $i < k$ the value of the shared variables is stored in q_{i+1}.

Function `linear_int` is defined with formal parameters $\overline{q} = \langle q_1, \dots, q_k \rangle$, $\overline{q}' = \langle q'_1, \dots, q'_{k-1} \rangle$ and *bound*. Variable *bound* stores the bound on the number of rounds to execute in the current call to `linear_int`.

The variable *atom* is set to true when entering an atomic block and set back to false on exiting it. The interlined code refers to variables *last* and j. The variable *last* is nondeterministically assigned when `linear_int` starts. Variable j counts the rounds being executed so far in the current call of `linear_int` (j is initialized to 1).

We also insert "`assume`(F);" before each return statement of `linear_int` which is not part of the interlined code; this prevents a call to `linear_int` to be returned after executing to completion.

Correctness and laziness of the sequentialization

We now formally prove the correctness and laziness of our sequentialization. We start with a lemma stating that function `linear_int` indeed computes linear interfaces of the parameterized program $\mathscr{P}$ (i.e. meets its post-condition).

Lemma 2 *Assume that* `linear_int` *when called with actual parameters* $u_1, \dots, u_k$, $v_1, \dots, v_{k-1}, i$ *terminates and returns. If* $\widehat{s}$ *is the valuation of the (global) variable* s *at return, then* $(\langle u_1, \dots, u_i \rangle, \langle v_1, \dots, v_{i-1}, \widehat{s} \rangle)$ *is a linear interface of* P. □

Consider a call to `linear_int` such that the precondition stated in page 41 holds. Using the above lemma we can show that the localized states from which we simulate the transition of $\mathscr{P}$ are discovered lazily, and that the program ensures that the precondition holds on recursive calls to `linear_int`.

Lemma 3 *Let* $(\langle v_0, v_1, \dots, v_{i-1} \rangle, \langle u_1, \dots, u_i \rangle)$ *be an initial linear interface. Consider a call to* `linear_int` *with actual parameters* $u_1, \dots, u_k$, $v_1, \dots, v_{k-1}, i$.

- *Consider a localized state reached during an execution of this call, and let a statement of* $\mathscr{P}$ *be simulated on this state. Then the localized state is reachable in some execution of* P.
- *Consider a recursive call to* `linear_int` *with parameters* $u'_1, \dots, u'_k$, $v_1, \dots, v_{k-1}, j$. *Then* $(\langle v_0, v_1, \dots, v_{j-1} \rangle, \langle u'_j, \dots, u'_j \rangle)$ *is a linear interface.* □

Note that whenever the function `main` calls `linear_int`, it satisfies the pre-condition for `linear_int`. This fact along with the above two lemmas establish the soundness and laziness of the sequentialization.

The following lemma captures the completeness argument:

Lemma 4 *Let* ρ *be a* k*-round execution of* $\mathscr{P}$. *Then there is a wrapped initial linear interface* $(\langle u_1, \dots, u_k \rangle, \langle u_2, \dots, u_k, v \rangle)$ *that* ρ *conforms to, and a terminating execution* ρ' *of* $\mathscr{P}_k^{lazy}$ *such that at the end of* ρ'*, the valuation of the variables* $\langle q_1, \dots, q_k, s \rangle$ *is* $\langle u_1, \dots, u_k, v \rangle$. *Furthermore, every localized state reached in* ρ *is also reached in* ρ'. □

Consolidating the above lemmas, we have:

Theorem 1 *Given* $k \in \mathbb{N}$ *and a parameterized program* $\mathscr{P}$*, an assertion is violated in a* k*-round execution of* $\mathscr{P}$ *if and only if an assertion is violated in an execution of* $\mathscr{P}_k^{lazy}$. *Moreover,* $\mathscr{P}_k^{lazy}$ *is lazy: if* $\mathscr{P}_k^{lazy}$ *simulates a statement of* $\mathscr{P}$ *on a localized state, then the localized state is reachable in* $\mathscr{P}$. □

4.2 Parameterized programs over finite data domains:

A sequential program with variables ranging over finite domains can be modeled as a pushdown system. Analogously, a parameterized program with variables ranging over finite domains can be modeled as a parameterized multi-stack pushdown system, i.e., a system composed of a finite number of pushdown systems sharing a portion of the control locations, which can be replicated in an arbitrary number of copies in each run. A *parameterized multi-stack pushdown system* $\mathscr{A}$ is thus a tuple $(S, S_0, \{A_i\}_{i=1}^n)$, where S is a finite set of shared locations, $S_0 \subseteq S$ is the set of the initial shared locations and for $i \in [1,n]$, with A_i is a standard pushdown system whose set of control locations is $S \times L_i$ for some finite set L_i. We omit a formal definition of the behaviors of $\mathscr{A}$ which can be easily derived from the semantics of parameterized programs given in Section 2, by considering that each $s \in S$ is the analogous of a shared state in the parameterized programs, a state of each A_i is the analogous of a local state of a process, and thus $(s,l) \in S \times L_i$ corresponds to a localized state.

Following the sequentialization construction given earlier in this section to construct the sequential program $\mathscr{P}_k^{lazy}$ from a parameterized program $\mathscr{P}$, we can construct from $\mathscr{A}$ a pushdown system $\mathscr{A}_k$ such that the reachability problem under k-round schedules in $\mathscr{A}$ can be reduced to the standard reachability problem in $\mathscr{A}_k$. Also, the number of locations of $\mathscr{A}_k$ is $O(\ell\, k^2\, |S|^{2k})$ and the number of transitions of $\mathscr{A}_k$ is $O(\ell\, d\, k^3\, |S|^{2k-1})$ where ℓ is $\sum_{i=1}^n |L_i|$ and d is the number of the transitions of $A_1, \ldots, A_n$.

Theorem 2 *Let $\mathscr{A}$ be a parameterized multi-stack pushdown system and $k \in \mathbb{N}$. Reachability up to k-round schedules in $\mathscr{A}$ reduces to reachability in $\mathscr{A}_k$. Moreover, the size of $\mathscr{A}_k$ is singly exponential in k and linear in the product of the number of locations and transitions of $\mathscr{A}$.* □

5 Conclusions and Future Work

We have given an assertion-preserving efficient sequentialization of parameterized concurrent programs under bounded round schedules.

An interesting future direction is to practically utilizing the sequentialization to analyze parameterized concurrent programs. For concurrent programs with a finite number of threads, *bounded-depth* verification using SMT solvers has worked well, especially using eager translations [14, 8]. However, since the sequentialization described in this paper introduces recursion even for bounded-depth concurrent programs, it would be hard to verify the resulting sequential program using SMT solvers. We believe that verifying the sequential program using abstract interpretation techniques that are context-sensitive would be an interesting future direction to pursue; in this context, the laziness of the translation presented here would help in maintaining the accuracy of the analysis.

Finally, sequentializations can also be used to subject parameterized programs to abstraction-based model-checking. It would be worthwhile to pursue under-approximation of static analysis of concurrent and parameterized programs (including data-flow and points-to analysis) using sequentializations.

References

[1] Mohamed Faouzi Atig, Ahmed Bouajjani & Shaz Qadeer (2009): *Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads*. In Stefan Kowalewski & Anna Philippou, editors: *TACAS*, *Lecture Notes in Computer Science* 5505, Springer, pp. 107–123. Available at `http://dx.doi.org/10.1007/978-3-642-00768-2_11`.

[2] Gérard Basler, Michele Mazzucchi, Thomas Wahl & Daniel Kroening (2009): *Symbolic Counter Abstraction for Concurrent Software*. In Bouajjani & Maler [3], pp. 64–78. Available at http://dx.doi.org/10.1007/978-3-642-02658-4_9.

[3] Ahmed Bouajjani & Oded Maler, editors (2009): *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. *Lecture Notes in Computer Science* 5643, Springer. Available at http://dx.doi.org/10.1007/978-3-642-02658-4.

[4] Ariel Cohen & Kedar S. Namjoshi (2008): *Local Proofs for Linear-Time Properties of Concurrent Programs*. In Aarti Gupta & Sharad Malik, editors: *CAV*, *Lecture Notes in Computer Science* 5123, Springer, pp. 149–161. Available at http://dx.doi.org/10.1007/978-3-540-70545-1_15.

[5] E. Allen Emerson & Vineet Kahlon (2004): *Parameterized Model Checking of Ring-Based Message Passing Systems*. In Jerzy Marcinkowski & Andrzej Tarlecki, editors: *CSL*, *Lecture Notes in Computer Science* 3210, Springer, pp. 325–339. Available at http://dx.doi.org/10.1007/978-3-540-30124-0_26.

[6] Michael Emmi, Shaz Qadeer & Zvonimir Rakamaric (2011): *Delay-bounded scheduling*. In Thomas Ball & Mooly Sagiv, editors: *POPL*, ACM, pp. 411–422. Available at http://doi.acm.org/10.1145/1926385.1926432.

[7] Pranav Garg & P. Madhusudan (2011): *Compositionality Entails Sequentializability*. In Parosh Aziz Abdulla & K. Rustan M. Leino, editors: *TACAS*, *Lecture Notes in Computer Science* 6605, Springer, pp. 26–40. Available at http://dx.doi.org/10.1007/978-3-642-19835-9_4.

[8] Naghmeh Ghafari, Alan J. Hu & Zvonimir Rakamaric (2010): *Context-Bounded Translations for Concurrent Software: An Empirical Evaluation*. In Jaco van de Pol & Michael Weber 0002, editors: *SPIN*, *Lecture Notes in Computer Science* 6349, Springer, pp. 227–244. Available at http://dx.doi.org/10.1007/978-3-642-16164-3_17.

[9] Yonit Kesten, Oded Maler, Monica Marcus, Amir Pnueli & Elad Shahar (1997): *Symbolic Model Checking with Rich ssertional Languages*. In Orna Grumberg, editor: *CAV*, *Lecture Notes in Computer Science* 1254, Springer, pp. 424–435.

[10] Yonit Kesten, Amir Pnueli, Elad Shahar & Lenore D. Zuck (2002): *Network Invariants in Action*. In Lubos Brim, Petr Jancar, Mojmír Kretínský & Antonín Kucera, editors: *CONCUR*, *Lecture Notes in Computer Science* 2421, Springer, pp. 101–115. Available at http://dx.doi.org/10.1007/3-540-45694-5_8.

[11] Salvatore La Torre, P. Madhusudan & Gennaro Parlato (2009): *Analyzing recursive programs using a fixed-point calculus*. In Michael Hind & Amer Diwan, editors: *PLDI*, ACM, pp. 211–222. Available at http://doi.acm.org/10.1145/1542476.1542500.

[12] Salvatore La Torre, P. Madhusudan & Gennaro Parlato (2009): *Reducing Context-Bounded Concurrent Reachability to Sequential Reachability*. In Bouajjani & Maler [3], pp. 477–492. Available at http://dx.doi.org/10.1007/978-3-642-02658-4_36.

[13] Salvatore La Torre, P. Madhusudan & Gennaro Parlato (2010): *Model-Checking Parameterized Concurrent Programs Using Linear Interfaces*. In Tayssir Touili, Byron Cook & Paul Jackson, editors: *CAV*, *Lecture Notes in Computer Science* 6174, Springer, pp. 629–644. Available at http://dx.doi.org/10.1007/978-3-642-14295-6_54.

[14] Shuvendu K. Lahiri, Shaz Qadeer & Zvonimir Rakamaric (2009): *Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers*. In Bouajjani & Maler [3], pp. 509–524. Available at http://dx.doi.org/10.1007/978-3-642-02658-4_38.

[15] Akash Lal & Thomas W. Reps (2009): *Reducing concurrent analysis under a context bound to sequential analysis*. *Formal Methods in System Design* 35(1), pp. 73–97. Available at http://dx.doi.org/10.1007/s10703-009-0078-9.

[16] Madanlal Musuvathi & Shaz Qadeer (2007): *Iterative context bounding for systematic testing of multithreaded programs*. In Jeanne Ferrante & Kathryn S. McKinley, editors: *PLDI*, ACM, pp. 446–455. Available at http://doi.acm.org/10.1145/1250734.1250785.

[17] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar & Iulian Neamtiu (2008): *Finding and Reproducing Heisenbugs in Concurrent Programs*. In Richard Draves & Robbert van Renesse, editors: *OSDI*, USENIX Association, pp. 267–280. Available at `http://www.usenix.org/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf`.

[18] Amir Pnueli, Jessie Xu & Lenore D. Zuck (2002): *Liveness with (0, 1, infty)-Counter Abstraction*. In Ed Brinksma & Kim Guldstrand Larsen, editors: *CAV*, *Lecture Notes in Computer Science* 2404, Springer, pp. 107–122. Available at `http://dx.doi.org/10.1007/3-540-45657-0_9`.

[19] Shaz Qadeer & Jakob Rehof (2005): *Context-Bounded Model Checking of Concurrent Software*. In Nicolas Halbwachs & Lenore D. Zuck, editors: *TACAS*, *Lecture Notes in Computer Science* 3440, Springer, pp. 93–107. Available at `http://dx.doi.org/10.1007/978-3-540-31980-1_7`.

[20] Shaz Qadeer & Dinghao Wu (2004): *KISS: keep it simple and sequential*. In William Pugh & Craig Chambers, editors: *PLDI*, ACM, pp. 14–24. Available at `http://doi.acm.org/10.1145/996841.996845`.

[21] Dejvuth Suwimonteerabuth, Javier Esparza & Stefan Schwoon (2008): *Symbolic Context-Bounded Analysis of Multithreaded Java Programs*. In Klaus Havelund, Rupak Majumdar & Jens Palsberg, editors: *SPIN*, *Lecture Notes in Computer Science* 5156, Springer, pp. 270–287. Available at `http://dx.doi.org/10.1007/978-3-540-85114-1_19`.