Model-checking Parameterized Concurrent Programs using Linear Interfaces ⋆

S. La Torre[1], P. Madhusudan[2], G. Parlato[2]

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

We dedicate this paper to the memory of Amir Pnueli.

Abstract. We consider the verification of parameterized Boolean programs— abstractions of shared-memory concurrent programs with an unbounded number of threads. We propose that such programs can be model-checked by iteratively considering the program under k-round schedules, for increasing values of k, using a novel compositional construct called *linear interfaces* that summarize the effect of a block of threads in a k-round schedule. We also develop a game-theoretic sound technique to show that k rounds of schedule suffice to explore the entire search-space, which allows us to prove a parameterized program entirely correct. We implement a symbolic model-checker, and report on experiments verifying parameterized predicate abstractions of Linux device drivers interacting with a kernel to show the efficacy of our technique.

1 Introduction

Parameterized concurrent programs are concurrent programs with an *unbounded number of threads*, executing similar code (or code chosen from a finite set of programs). In the model-checking literature, parameterized programs have been heavily investigated (see section of related work), as they are a natural extension of concurrent systems, and a very relevant model for communication protocols and distributed systems. Model-checking parameterized programs, even when the data domain is finite, is, in general, *undecidable*.

In this paper, we propose a new technique to verify parameterized finite-data-domain programs, or parameterized Boolean programs. The primary idea is to *iterate* over k-round schedules of the parameterized program, for increasing values of k, and detect termination by proving that all reachable configurations have been reached at the k-th round, for some k.

More precisely, we work through phases, each phase for an increasing value of k, and model-check if the parameterized program can reach the error state, for some instantiation of n threads and in some k-round schedule. This task,

⋆ This work was partially funded by the MIUR grants FARB 2008-2009 Università degli Studi di Salerno (Italy), the NSF CAREER award #0747041, and the NSF Award #0917229. This is a technical report of a conference paper published in the proceedings of Computer Aided Verification (CAV'10), Lecture Notes in Computer Science, Edinburgh, UK, 2010, Springer.

though an under-approximation of the reachable state-space, is challenging, as the number of threads is not fixed. We develop a novel construct, called *linear interfaces*, that summarizes the effect of an *arbitrary* block of threads in a k-round schedule. Linear-interfaces (as opposed to general interfaces), capture the effect of a block of threads along a *single run* that context-switches into and out of the block.

The lack of branching information and the finite description of linear interfaces helps us to build a compositional framework to search the state space, that combines linear interfaces without blow-up. We develop a fairly intricate algorithm that uses linear interfaces for blocks of threads scheduled at the right end of each round (*right-blocks*), to ensure that we never leave the set of reachable states in constructing linear interfaces. Further, the algorithm can be captured as a fixed-point computation over an appropriate signature, and hence naturally yields to symbolic BDD-based methods.

Our second contribution is an *adequacy check* that tries to prove that all reachable states of a parameterized program are already reached under some k-round schedule. This check, which is sound but not complete, is formulated as a two-player reachability game on an (implicitly defined) graph. Intuitively, Eve (player 0) aims to show that there is a global state reachable in the $(k+1)$-th round that is not reachable in the k-th round, and Adam (player 1) aims to disprove this. The game works by Eve *declaring* a global state, by declaring one at a time the local states on each thread, and Adam responds by reaching the same states using only k rounds. If Adam has a *winning strategy* (and hence Eve has none), then this proves that every global state reachable in the $(k+1)$-th round is already reachable in the k-th round. Thus, we can stop computing for higher values of k and declare the program correct. The idea of formulating the check as a *game* is a technical novelty, and is used to declare a state that involves an *arbitrary large* number of threads step by step (she cannot very well declare the global state in one stroke as then the game-graph will no longer be finite). However, the fact that Eve declares the global state one thread at a time can give her an advantage in the game, and if Eve has a winning strategy, we cannot conclude that a configuration is reachable in the $(k+1)$-th round and not in the k-th round. Hence our adequacy check is sound but incomplete. The game, and finding whether Adam has a winning strategy (i.e. solving the game), can also be formulated and computed symbolically.

The idea of slicing the reachable state-space in terms of the number of rounds is non-traditional (classic approaches would induct over the number of threads) and is motivated by recent work on slicing the state-spaces of concurrent programs using a bounded number of context switches. Bounded context-switching is motivated by the belief that most errors (and, in fact, most reachable states) will be already reachable in a few number of rounds [20]. Also, from an algorithmic perspective, model-checking under k-round schedules is *decidable* and can be achieved using, at any point, only *one* copy of the local state of a thread, and $O(k)$ copies of the shared variables.

Our work argues that the above can be exploited also for *parameterized* systems, thus obtaining an effective decidable way of exploring search spaces. Moreover, our *adequacy check*, which is entirely novel, can verify (soundly) that searching beyond k-round schedules is useless, and hence terminate the search, proving the parameterized program correct for any number of threads and any schedule. We emphasize that the completeness check closely follows and relies on the bounded-round schedule reachability algorithm.

While several approaches in the literature have explored bounded context-switching as an *under-approximation* to find errors, to our knowledge ours is the first to use this under-approximation to prove that the program is in fact entirely correct. Our adequacy check works for parameterized programs, but no similar check is known even for concurrent programs with finitely many threads. We thus believe that the analysis of such programs would benefit from using it.

We report on a symbolic BDD-based implementation of both the k-round model-checking for parameterized programs as well as the k-round adequacy check. Our implementation is a succinct formulation of the algorithms using fixed-points, and we use the GETAFIX framework [13] that we have developed recently, to implement our algorithm by simply writing fixed-point equations.

We report on using our model-checker to verify a large suite of Boolean parameterized programs obtained from the DDVERIFY tool, that extracts Boolean models of Linux device drivers and the OS kernel, using predicate abstraction, in order to check them against rules of kernel API usage (similar to SLAM, which is for Windows drivers). Our parameterized setting models an arbitrary number of these drivers working with the OS. We report on experiments performed on about 8500 programs and properties, and show that our tool can effectively find reachable error-states, and furthermore *prove* that more than 80% of them are entirely correct, using the adequacy check.

In summary, our theoretical and experimental results suggest a new technique for verifying parameterized programs: to effectively under-approximate them using a few round schedules (but with arbitrary number of threads), summarized and analyzed using linear interfaces, and build effective techniques to prove a few rounds suffice to reach the entire reachable state-space.

Details of the implementation of the idea presented in this paper is at the GETAFIX website: `http://www.cs.uiuc.edu/~madhu/getafix`.

Related work. Compositional verification using interfaces for modules has been investigated before: e.g. the work in [4] computes interfaces for modules using *learning* for compositional verification. However, these interfaces are modeled as *finite transition systems*, and will not help in verifying unboundedly many threads as the interfaces, when composed, will keep increasing in size.

The idea of exploring search-spaces of concurrent programs with finitely many threads, using a small number of context-switches for finding bugs has been well studied recently [20, 17, 19, 21, 16, 13, 14]. The CHESS tool from Microsoft espouses this philosophy by testing concurrent programs by systematically choosing schedules with a small number of context-switches/pre-emptions.

A recent paper [1] proposes a (theoretical) solution to the model-checking problem of reachability in concurrent programs with dynamic creation of threads, where a thread is context-switched into only a bounded number of times. This dynamic thread creation can model the unboundedly many threads in our setting. However, dynamic thread creation requires keeping track of the *number of threads that are in a local state*, even under bounded switching. The paper in fact shows 2-way reductions between this reachability problem and Petri-net coverability, establishing Expspace-completeness. In contrast, it follows from our fixed-point formulation that the model-checking problem in our setting is Pspace-complete. More importantly, our fixed-point formulation actually yields a practical *symbolic BDD-based solution*, while it is not clear how to build a symbolic model-checker using the Petri-net reduction given in [1] (the paper does not report any implementation or experiments).

There is a rich history of verifying parameterized asynchronously communicating concurrent programs, especially motivated by the verification of distributed protocols: sample research includes network invariants (see [12] and references therein) and its abstractions [3, 10, 6]; *regular model-checking* [11], using small-model theorems [7]; *split invariants* followed by abstractions based on this invariant and model-checking [5]. Symmetry in replicated concurrent processes [8] has been exploited in the Murφ tool [10].

Approaches for verifying several replicated components (though finite) have used *counter abstraction* [18], and recent work has used counter abstraction combined with cartesian representations of local and global state in order to verify a fixed number of Linux device drivers working in paralell [2]. The model-checking work we report in this paper handles the same device drivers but with an *unbounded number* of them working in parallel and restricted to a bounded number of round schedules.

Predicate abstraction for parameterized systems have also been investigated where the predicates capture global invariants with index variables [15]; methods using abstract-interpretation techniques over standard abstraction domains have also been investigated [9].

2 Parameterized Boolean programs

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

Each parameterized program consists of a sequential block of statements `init`, where the shared variables are initialized, and a list of concurrent processes. Each *process* is essentially a sequential program (namely, a Boolean program) with explicit syntax for nondeterminism and (recursive) function calls, along with the possibility of declaring sets of statements to be executed *atomically*. Functions are all call-by-value. Variables can be scoped locally to a function, globally to a process in a thread or shared amongst all processes in all threads. The statements in a parameterized program can refer to all variables in scope.

A parameterized program is initialized with an arbitrary finite number of threads, each thread running a copy of one process. Dynamic creation of threads is not allowed, but it can be modeled by having the threads in a "dormant" state until a message from the parent thread is received.[3]

An *execution* of a parameterized program is obtained by interleaving the behaviors of the threads which are involved in it. For a concurrent process we assume the standard semantics of sequential programs (the request of executing atomically a block of statements has no meaning when executing a single thread). Formally, let $\mathcal{P} = (S, \texttt{init}, \{P_i\}_{i=1}^n)$ be a *parameterized program* where S is the set of shared variables and P_i is a process, $i \in [1, n]$. We assume that each statement of the program has a unique *program counter* labeling it. A *thread* T of $\mathcal{P}$ is a copy (instance) of some P_i, $i \in [1, n]$. At any point, only one thread is *active*. For any $m > 0$, a *state* of $\mathcal{P}$ 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 currently active thread is T_i, $i \in [1, m]$, (3) s is a valuation of the shared variables, and (4) for each $j \in [1, m]$, σ_j is a *local state* of T_j. Observe that each such σ_j is composed of a valuation of the program counter, and of the local and global variables of the corresponding process, along with a *call-stack* of local variable valuations and program counters to model function 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 when a thread is not scheduled, its local state 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.

Reachability. Given a parameterized program $\mathcal{P} = (S, \texttt{init}, \{P_i\}_{i=1}^n)$ and a target program counter pc, the *reachability problem* asks whether there exist an integer $m > 0$ and an execution of $\mathcal{P}$ that reaches a state $(map, i, \nu, \sigma_1, \ldots, \sigma_m)$ such that pc is the program counter of σ_i for some $i \in [1, m]$. Since two threads communicating through a finite shared memory suffice to simulate a Turing machine, this problem is clearly undecidable. Here we also consider the reachability under bounded-round schedules. For threads $T_1, \ldots, T_m$, a k-round schedule of $T_1, \ldots, T_m$ is a schedule that, for some ordering of such threads, activates them in k rounds, where in each round each thread is scheduled (for any number of events) according to this order. Observe that, restricting to executions under any

[3] Note: in this scheme, each thread creation causes a context-switch; true thread creation, without paying such cost (like in [1]), cannot be modeled in our framework.

k-round schedule does not place any bound on the number of threads which are involved. Given a $k \in \mathbb{N}$, the *reachability problem under bounded-round schedules* is the reachability problem restricted to consider only executions under k-round schedules.

3 Linear interfaces

We now introduce the concept of linear interface, that captures the effect a block of threads has on the shared state, when involved in an execution of a k-round schedule.

In the rest of the paper, we fix a parameterized program $\mathcal{P} = (S, \texttt{init}, \{P_i\}_{i=1}^n)$ and a bound $k > 0$ on the number of rounds. We also use the notation $\overline{u}$ to refer to a tuple $(u_1, \ldots, u_k)$ of shared states of $\mathcal{P}$.

A *linear interface* of length k is a pair $(\overline{u}, \overline{v})$ of tuples of k shared states, such that there is an execution of some ordered block of threads $T_1, \ldots, T_m$ of $\mathcal{P}$ where in k rounds, for $i = 1, \ldots, k$, when starting the first thread in the shared state u_i, the round ends in state v_i. Note that this execution within the block must preserve the local state of threads across consecutive rounds. In the following, we will often refer to $\overline{u}$ as the *input* and $\overline{v}$ as the *output* of $(\overline{u}, \overline{v})$.

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

Definition 1. *(*Linear interface*) 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 $\mathcal{P}$ (with processes P). The pair $(\overline{u}, \overline{v})$ is a* linear interface *of $\mathcal{P}$ 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 $\mathcal{P}$ for $i \in [1, m]$ and $j \in [1, k]$, such that, for each $i \in [1, m]$ and $j \in [1, k]$:*

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

When $m = 1$, $(\overline{u}, \overline{v})$ is also called a thread linear interface*.* □

Note that the definition of a linear interface $(\overline{u}, \overline{v})$ 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 $\overline{u}$ as input and compute $\overline{v}$ in the k rounds, preserving the local configuration of threads between rounds.

Linear interfaces compose. Let $I = (\overline{u}, \overline{v})$ and $I' = (\overline{u}', \overline{v}')$ be two linear interfaces of length k. If the output of I matches the input of I', i.e., $\overline{v} = \overline{u}'$ holds, then the *composition* of I and I' is the pair $(\overline{u}, \overline{v}')$.

Lemma 1. *The composition of linear interfaces of length k is a linear interface of length k. Moreover, each linear interface is either a thread linear interface or a composition of two or more thread linear interfaces.* □

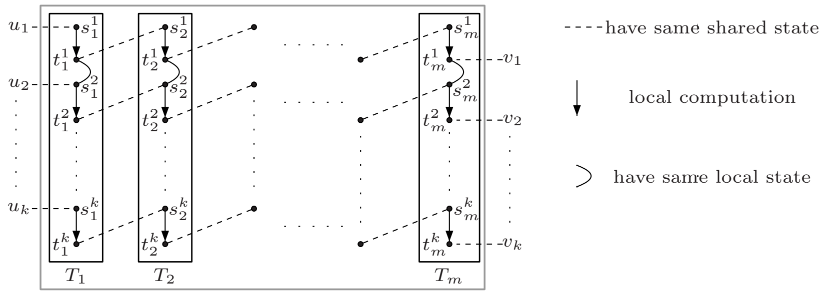

Fig. 1. A linear interface

An execution of a parameterized program under a k-round schedule can always be seen as a composition of thread linear interfaces that form a unique linear interface that have the following properties.

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

Thus, an execution of a parameterized program under a k-round schedule always corresponds to a wrapped initial linear interface $(\overline{u}, \overline{v})$. Such an execution is said to *conform* to $(\overline{u}, \overline{v})$. The following lemma is straightforward:

Lemma 2. *Let $\mathcal{P}$ be a parameterized program. An execution of $\mathcal{P}$ is under a k-round schedule iff it conforms to some wrapped initial linear interface of $\mathcal{P}$ of length k.* $\square$

4 Reachability under bounded-round schedules

In this section we give a fixed-point algorithm to solve the reachability problem under a bounded-round schedule for a parameterized program. From Lemma 2, it follows that all that is required is to compute, for a given parameterized program, all possible linear interfaces of size k, and then check among those that are both initial and wrapped. Since for a fixed k the number of linear interfaces of a program is finite, this can be computed as suggested by Lemma 1, starting with thread linear interfaces, and then composing them till a fixed-point is reached. However, it turns out that this does not work well in practice, as the computation of thread linear interfaces starts from arbitrary tuples of k shared states and then determines all the states reachable from them, and hence unreachable parts of the state-space can be explored. Early implementation results of this algorithm in fact failed miserably on our benchmarks. We now propose a more intricate algorithm that ensures that linear interfaces are computed and explored only on reachable states.

Notation: Let π be an execution of $\mathcal{P}$ under a k-round schedule and $T_1, \ldots, T_m$ denote a block of threads scheduled consecutively in π. We say that π *covers* a linear interface $(\overline{u}, \overline{v})$ on $T_1, \ldots, T_m$ if along π, u_i matches the shared state on

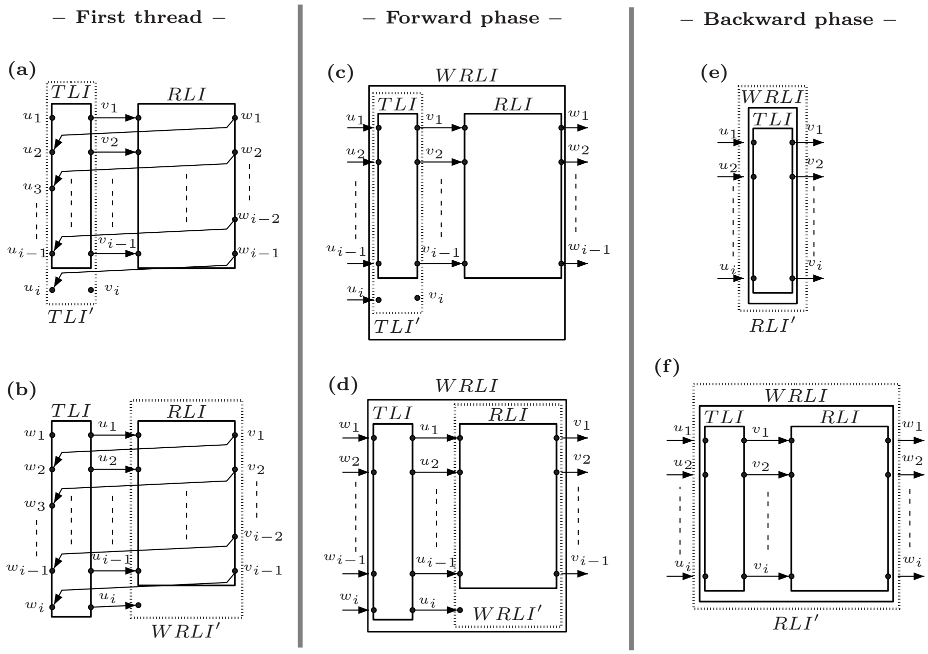

Fig. 2. Graphical representation of the update rules of the algorithm.

context-switching into T_1 and v_i matches the shared state on context-switching out of T_m in round i, for $i \in [1, k]$. Moreover, the localized state (σ, v_k) of T_m, which is visited along π when context-switching out of T_m in round k, is called a *final localized state* of $(\overline{u}, \overline{v})$ in π. A *right block* is a block of threads scheduled consecutively in the end of each round.

Description of the algorithm. The algorithm proceeds by computing for the input program, linear interfaces of size $1, 2, \ldots k$, ensuring that each is computed on reachable states only. In every iteration, we also compute the precise set of linear interfaces for right blocks (*right linear interfaces*).

Let us now describe, intuitively, how the i-th round is explored and how interfaces of length i are built when $i > 1$. We refer the reader to the diagrams in Fig. 2. In these diagrams, boxes drawn with solid lines denote interfaces that exist, while those with dotted lines denote new blocks that get created. Moreover, *TLI*, *RLI*, and *WRLI* refer to thread-linear interfaces, right linear interfaces, and "want blocks". Arrows denote equality of the shared states at the endpoints.

We start the i-th round with the first thread (see Fig. 2.a). We take an initial thread linear interface (TLI) of length $i-1$ and a right linear interface (RLI), still of length $i-1$, which composes with TLI and such that the resulting linear interface is both initial and wrapped. We then compute a localized state (σ, u_i), where σ is from the final localized state of TLI and u_i is the shared state from the end of the $(i-1)$-th round in *RLI*. Using this, we can compute all possible states which are reachable by the thread in round i, and hence compute all the thread linear interfaces of length i covered by a run on the first thread (Fig. 2.a). Now the computation progresses on the second thread (see Fig. 2.b). For all the newly reached shared states u_i, we then create a *want block* ($WRLI'$) with the RLI's input, the new input u_i, and the RLI's output, which captures our desire that we want to continue the rest of the threads with this new input. Want blocks are not quite linear interfaces, as they have i inputs and $i-1$ outputs, but are crucial in guiding the computation.

Next, we enter the *forward phase* (Fig. 2.c and 2.d), where a want-block $WRLI$, a thread linear interface TLI, and a right linear interface RLI exist, and where the inputs of $WRLI$ and TLI match, the outputs of TLI match the input of RLI, and the outputs of $WRLI$ and RLI match. In this scenario, a new thread linear interface of size i is formed from TLI (inheriting the shared state from $WRLI$ and the local state from TLI) and explored locally to form new thread linear interfaces of size i (Fig. 2.c). Further, these new thread linear interfaces create further want blocks to further the computation (Fig. 2.d).

Want blocks can also (non-deterministically) stop when the inputs precisely match the outputs, and create right linear interfaces (Fig. 2.e). This is the base case of the induction capturing the formation of right linear interfaces (starting from the last scheduled thread in each round) and starts the *backward phase*. This computation takes a right linear interface RLI, combines it with a thread linear interface TLI to the left of it, and provided a matching want block exists, combines them to form a larger right linear interface (Fig. 2.f). These computed right linear interfaces correspond to reachable blocks of computation (because we have checked them against want blocks, which were in turn reachable), and is used in the next iteration to ensure that only reachable states are explored.

Of course, the above three phases are not regulated sequentially, and are explored arbitrarily by fixed-point computations.

Fixed-point formulation. We formally describe our algorithm as a system of equations of the form $R = Exp$ where Exp is a positive boolean expression with first order quantification over relations and R is a relation which may also appear within Exp (*recursive* definition of relations is admitted).

In such equations, we will use the following base relations. $LocInit$ and $ShInit$ denote respectively the initial local states for each thread and the initial shared states (computed by executing the `init` block). $Wrap(\overline{u}, \overline{v})$ holds true if and only if $v_i = u_{i+1}$, for all $i \in [1, k-1]$. We also use $\langle$ *local reachability* $\rangle$ to denote a formula expressing the clauses of a fixed-point formulation of the states that are forward reachable using only transitions of a process. We omit the details on this formula since it is essentially the same as for sequential programs (see [13]).

Denote with $\mathcal{S}$ the following system of equations:

1. $TLI(i, \sigma, \overline{u}, \overline{v}) =$

$$(i = 1 \wedge LocInit(\sigma) \wedge u_1 = v_1 \wedge (ShInit(u_1) \vee \exists\, \overline{w}, \sigma'.\, TLI(1, \sigma', \overline{w}, \overline{u}))) \quad (1.1)$$

$$\vee\, (i > 1 \wedge u_i = v_i \wedge TLI(i-1, \sigma, \overline{u}, \overline{v}) \wedge \exists\, \overline{w}.\ (RLI(i-1, \overline{v}, \overline{w}) \wedge (\ (ShInit(u_1) \wedge Wrap(\overline{u}, \overline{w})) \vee WRLI(i, \overline{u}, \overline{w})))) \quad (1.2)$$

$$\vee\ \langle\, local\ reachability \,\rangle \quad (1.3)$$

2. $\mathit{WRLI}(i, \overline{u}, \overline{v}) = \quad i > 1 \wedge \exists \sigma, \overline{w}.$
$(\mathit{TLI}(i, \sigma, \overline{w}, \overline{u}) \wedge \mathit{RLI}(i-1, \overline{u}, \overline{v}) \wedge (\mathit{WRLI}(i, \overline{w}, \overline{v}) \vee (\mathit{ShInit}(w_1) \wedge \mathit{Wrap}(\overline{w}, \overline{v}))))$

3. $\mathit{RLI}(i, \overline{u}, \overline{v}) = \quad (i = 1 \vee \mathit{WRLI}(i, \overline{u}, \overline{v}))$
$\wedge \exists \sigma. (\mathit{TLI}(i, \sigma, \overline{u}, \overline{v}) \vee (\exists \overline{w}.\ (\mathit{TLI}(i, \sigma, \overline{u}, \overline{w}) \wedge \mathit{RLI}(i, \overline{w}, \overline{v})))$

Observe that $\mathcal{S}$ is a system of positive equations. Thus by Tarski's fixed-point theorem, it has a unique least fixed-point, and the relations are well defined. The evaluation of $\mathcal{S}$ is graphically described in Fig. 2.

After computing the above relations, the last step of our algorithm consists of evaluating the formula:

$$\varphi ::= \exists i, \sigma, \overline{u}, \overline{v}. (1 \leq i \leq k) \wedge \mathit{TLI}(i, \sigma, \overline{u}, \overline{v}) \wedge \mathit{Target}(\sigma),$$

where the predicate $\mathit{Target}(\sigma)$ holds if and only if σ corresponds to a target program counter in the reachability query.

Correctness of the algorithm. The following lemma is crucial to prove our algorithm correct.

Lemma 3. *Let $\overline{u} = (u_1, \ldots, u_k)$, $\overline{v} = (v_1, \ldots, v_k)$, σ such that (σ, v_i) is a localized state of $\mathcal{P}$, $k \in \mathbb{N}$, and $i \leq k$.*

1. *$\mathit{TLI}(i, \sigma, \overline{u}, \overline{v})$ holds iff there is an execution π of $\mathcal{P}$ under a k-round schedule such that $(\overline{u}_i, \overline{v}_i)$ is a thread linear interface covered by π and (σ, v_i) is a final localized state of $(\overline{u}_i, \overline{v}_i)$.*
2. *$\mathit{RLI}(i, \overline{u}, \overline{v})$ holds iff there is an execution π of $\mathcal{P}$ under a k-round schedule such that $(\overline{u}_i, \overline{v}_i)$ is a right linear interface covered by π.*
3. *$\mathit{WRLI}(i, \overline{u}, \overline{v})$ holds iff there is an execution π of $\mathcal{P}$ under a k-round schedule such that $T_1, \ldots, T_m$ are scheduled at the end of each round, u_i is the shared state on context-switching to T_1 along π in round i, $i > 1$, and $(\overline{u}_{i-1}, \overline{v}_{i-1})$ is a right linear interface covered by π on $T_1, \ldots, T_m$.* □

Note that, when computing the fixed point of $\mathcal{S}$, the relations *TLI*, *RLI* and *WRLI* grow monotonically, and once a tuple is added, it is never removed from the set. Thus, from the lemma, we get that in our computation, we only explore the reachable state space of the parameterized program. Therefore, we have:

Theorem 1. *Given an integer $k \geq 0$, a parameterized program $\mathcal{P}$ and a program counter pc, pc is reachable in $\mathcal{P}$ under k-round schedules if and only if the formula φ is satisfiable. Moreover, while computing the least fixed-point of system $\mathcal{S}$, only reachable localized states of $\mathcal{P}$ are explored.* □

5 An adequacy check: proving program correct

The algorithm to solve the reachability problem under a k-round scheduling, given in the previous section, can be used to show a parameterized program incorrect (when an error state is reached). However, when the algorithm's answer is negative (i.e., an error state is not reachable) nothing can be inferred on the correctness of the input program. In this section, we present an *adequacy check* that attempts to make our verification scheme complete. In particular, for a parameterized program without recursive function calls, we design a test that gives a sufficient condition to show that the reachable states of the program under a k-round schedule are indeed all its reachable states. Though the proposed test is sound but incomplete, in next section, we show by reporting our experimental results that it is indeed quite effective in practice.

Fix a parameterized program $\mathcal{P}$ and $k \in \mathbb{N}$. We wish to ensure the following: "For any state s of $\mathcal{P}$, if s is reachable under a $(k+1)$-round schedule then it is also reachable under a k-round schedule" (*k-rounds-suffice condition*).

Note that checking this condition may be computationally hard, and hardness mostly resides in the fact that the number of threads in the executions under k-round schedules is a priori unbounded (and thus handling entire program states is by itself a problem). We propose a game-theoretic algorithm that essentially refers to portions of states which are local to threads (localized states) and keeps summaries of the performed computation (linear interfaces), and thus does not need to refer to the entire state of the program, but rather parses it thread-by-thread.

In particular, we wish to define a two-player game G_k where player 0 (*Eve*) selects a state of $\mathcal{P}$ by revealing with each move a localized state which is visited along an execution under a $(k+1)$-round schedule in round $k+1$, and player 1 (*Adam*) attempts to match every move of Eve along an execution under the same schedule but in round k. A typical play in G_k is as follows.

Eve starts selecting a localized state λ_1 which is final for an initial thread linear interface I_1 of length $k+1$ (we recall that this means that there exists a program execution under a k-round schedule which covers I_1 and context-switch out of the first thread in round $k+1$ at λ_1). Then, Adam matches this move by showing that λ_1 is a final localized state of an initial thread linear interface L_1 of length k. The play continues with Eve selecting a final localized state λ_2 of a thread linear interface I_2 of length $k+1$ such that the output of I_1 matches the input of I_2. Then, Adam reacts by showing that λ_2 is also a final localized state of a thread linear interface L_2 of length k such that the output of L_1 matches the input of L_2. Let I'_2 be the composition of I_1 and I_2, and L'_2 be the composition of L_1 and L_2. In the next iteration, Eve makes a selection expanding over the next thread in the schedule the linear interface I'_2, and similarly, Adam tries to matches this selection by expanding L'_2, and so on until Adam cannot match a move of Eve. Then starting from this point till the end, only Eve is allowed to move and she will keep expanding the constructed linear interface as above.

A play is winning for Eve if she can select a sequence of moves that cannot be matched by Adam and doing so she can construct a wrapped and initial linear interface, thus proving that the selected localized states are indeed visited in the $(k+1)$-th round of an execution under a $(k+1)$-round schedule. Eve also wins if Adam matches all her moves, but the linear interface she constructs is wrapped while that by Adam is not. In all the other cases, Adam wins.

Technically, we can store in the states of the game the interfaces which are constructed by the two players and thus express such winning conditions as reachability goals. Also note, that fixing k, the size of G_k is bounded.

We can formally describe a decision algorithm to solve such a game using equations as in Section 4. In our formulation, we model a state of the game as a tuple of the form $s = (pl, in, al, \overline{u}, \overline{v}, \sigma, \overline{x}, \overline{y})$ where pl denotes the player which is in control of the state (0 for Eve and 1 for Adam), $in = 1$ iff player pl has not moved yet in the current play, $al = 1$ iff Adam is still in the play (i.e., he has matched all Eve's moves so far), $(\overline{u}, \overline{v})$ is the linear interface of length $k+1$ constructed by Eve in the play, (σ, v_{k+1}) is a final localized state of $(\overline{u}, \overline{v})$, and $(\overline{x}, \overline{y})$ is the linear interface of length k constructed by Adam.

The winning conditions can be captured with a simple predicate characterizing the winning states. To solve the game, the attractor-set based algorithm can be easily expressed using fixed points and therefore we can directly implement it in our formalism. Due to the lack of space, we only give here the details of the relation *E-move* which captures the moves of Eve (the relation for Adam being similar).

$$
\begin{aligned}
&\mathit{E\text{-}move}(s, s') = \ (\ pl = 0 \wedge \overline{x}' = \overline{x} \wedge \overline{y}' = \overline{y} \wedge (\\
&\quad (al = 1 \wedge pl' = 1 \wedge al' = 1 \wedge \\
&\qquad ((in = 1 \wedge in' = 1 \wedge \mathit{TLI}(k+1, \sigma, \overline{u}', \overline{v}') \wedge \mathit{ShInit}(u_1)) && (1)\\
&\qquad\ \vee (in = 0 \wedge in' = 0 \wedge \overline{u}' = \overline{u} \wedge \mathit{TLI}(k+1, \sigma', \overline{v}, \overline{v}')))) && (2)\\
&\quad \vee (al = 0 \wedge pl' = 0 \wedge al' = 0 \wedge in = 0 \wedge \overline{u}' = \overline{u} \wedge \mathit{TLI}(k+1, \sigma', \overline{v}, \overline{v}')))) && (3)
\end{aligned}
$$

In the above formula, (1) corresponds to the first move of Eve in a play, (2) to her moves as long as Adam has matched all her previous moves, and (3) to her moves in the remaining cases (i.e., Adam has failed to match a move by Eve).

Observe that, if we restrict to parameterized programs where only non-recursive function calls are allowed, we can prove that if there is a winning strategy of Adam then the k-rounds-suffice condition holds, and therefore, there are no more reachable states to explore. However, the converse does not hold: if Eve has a winning strategy, we cannot conclude that considering executions under $(k+1)$-round schedules will allow us to discover new reachable states of the program. In fact, Eve could cheat by changing her selections depending on Adam's moves, and thus, even if a selected state is reachable within k rounds, Adam could fail to prove it. Thus, we have the following theorem:

Theorem 2. *Let $\mathcal{P}$ a parameterized program without recursive function calls. For all $k \in \mathbb{N}$, if the adequacy check holds then the k-rounds-suffice condition holds, and therefore all reachable states of $\mathcal{P}$ are visited in executions under k-round schedules.* □

6 Implementation and experiments

Symbolic model-checker: We implemented a symbolic BDD-based model-checker for reachability in parameterized programs in a bounded number of rounds, as well as a symbolic *adequacy checker* that checks (soundly) whether k-round schedules reach all reachable states, using the tool framework Getafix [13]

		2 thread Analysis		Parameterized Analysis 4 rounds			Parameterized Analysis unbounded rounds		
	#Bool. pgms.	Reach-able	Unreach-able	Reachable	Unreach-able	Time-out	Proved Unreach-able	Not proved unreachable (Pl.0 wins)	Time-out
i8xx_tco	765	460	305	314 (+13)	218	220	204	0	14
ib700wdt	492	330	162	208 (+13)	112	159	106	0	6
machzwd	568	341	227	274 (+23)	158	113	56	87	15
mixcomwd	429	276	153	213 (+23)	102	91	100	0	2
pcwd	256	171	85	171 (+0)	85	0	81	0	4
sbc60xxwdt	425	276	149	174 (+23)	94	134	92	0	2
sc1200wdt	491	299	192	200 (+13)	135	143	135	0	0
sc520_wdt	438	272	166	173 (+23)	104	138	15	89	0
smsc37b787_wdt	719	428	291	280 (+13)	140	286	140	0	0
w83877f_wdt	558	362	196	219 (+23)	103	213	15	88	0
w83977f_wdt	850	495	355	366 (+13)	126	345	125	0	1
wdt977	799	486	313	338 (+13)	127	321	125	0	2
wdt	533	348	185	221 (+17)	107	188	105	0	2
wdt_pci	892	800	92	378 (+23)	13	478	10	3	0
Total	**8215**	**5344**	**2871**	**3529 (+233)**	**1624**	**2829**	**1309**	**267**	**48**

Table 1. Experimental results.

that we have recently developed. Getafix allows writing BDD-based model-checkers using a high-level fixed-point calculus, without having to write low-level code. GETAFIX translates Boolean programs to logical formulas, implements heuristics for BDD orderings, and furnishes the model-checker designer with templates that capture the semantics of the program. High-level model-checking algorithms written in a fixed-point calculus get implemented by GETAFIX using the symbolic fixed-point model-checker called MUCKE. We refer the reader to the paper [13] for details on GETAFIX.

We adapted GETAFIX to translate parameterized Boolean programs and handle DDVERIFY benchmarks. The algorithms for reachability in k rounds were implemented using the fixed-point formulas outlined in this paper. The adequacy check was also implemented using fixed-points: we captured the moves of player 0 and player 1 symbolically, and wrote a fixed-point backward attractor-based algorithm to solve the reachability game.

Experiments on device drivers: We subject our parameterized model-checker to a suite of Boolean programs derived from the DDVERIFY tool [22], which abstracts Boolean programs from Linux device drivers, and also provides a fairly accurate Boolean model of the OS kernel. The model of driver is obtained using predicate abstraction, and appropriate translations of Spinlocks, timer functions, and service routines that it may use. The kernel program models kernel code as well as other OS related behavior such as interrupts, etc. using non-determinism.

Each DDVERIFY benchmark consists of a kernel module that interacts with a device driver. We obtained our concurrent models by taking *one* copy of the kernel module along with an *unbounded* number of copies of the device driver module. We subject our tool to about 8000 Boolean abstractions of 14 device drivers, abstracted to verify several (hundreds of) safety properties, at various levels of refinement.

The results are summarized in Table 1. The "2-thread analysis" columns report the number of Boolean programs that had an error-state reachable and those that did not, when considering just two threads, one modeling the OS and one modeling the driver (these results are identical to DDVerify).

We analyzed the programs using our parameterized analysis tool and searched the space reached within 4-round schedules for errors; the results are reported in the second set of columns. Note that even when an error state is reachable in the 2-thread analysis, it may not be reachable in the parameterized analysis (as the latter considers only a limited number of rounds); however this *never occurred* in our experiments. Similarly, note that when an error state is unreachable in the 2-thread analysis, it may be reachable in the parameterized analysis (as the the latter considers an unbounded number of threads); this did happen in several examples, and is noted in parenthesis with a +-sign in the "Reach" column of the parameterized analysis (e.g., for the first set of drivers, the error state was reachable in 13 programs in the parameterized setting within 4 rounds, but not in the 2-thread setting). The parameterized analysis is computationally more expensive, and the model-checker ran out of resources (memory or time-out at 30sec) for the programs reported in the "Timeout" column.

The final set of columns report results for the *adequacy check* based on the reachability game on those programs that were unreachable in 4 rounds. The first column reports the number of programs our tool was able to prove entirely correct (any number of rounds and threads); the second column reports the number of programs that were not proved unreachable (this does not mean that the error state *is* reachable, as our adequacy check is not complete); and the last column gives the programs on which the tool ran out of resources (out of memory or reached time-out at 30sec). For example, in the first set of drivers, out of the 218 programs in which the error state was not reachable in 4 rounds, our tool was able to prove 204 of them completely correct, and 14 of them timed-out.

Observations from experiments: Several observations are in order:

- All error-states reachable in the 2-thread instantiation were found within 4 rounds in the parameterized system. This experimentally supports the conjecture that error-states are often reachable within a few rounds, even on Boolean program abstractions.
- There are several programs (~ 225) where a predicate abstraction that can prove a driver correct when working alone with the OS is not sufficient to prove it correct in the parameterized setting.
- Most interestingly, most programs (~ 1300 out of 1600, or $\sim 80\%$), when the error state was not reachable in 4 rounds, were proved entirely correct by our technique. In fact, our adequacy check was extremely effective in 11 of the 14 suites; 3 suites however have a significant percentage of programs that we were unable to prove entirely correct.

Note that a sound predicate abstraction followed by a successful parameterized verification proves the original driver correct for any number of threads and schedule; our tool achieves this for about 1300 instances.

References

1. M. F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. In *TACAS*, vol. 5505 of *LNCS*, pp. 107–123. Springer, 2009.
2. G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. Symbolic counter abstraction for concurrent software. In *CAV*, vol. 5643 of *LNCS*, pp. 64–78. Springer, 2009.
3. E. M. Clarke, O. Grumberg, and S. Jha. Veryfying parameterized networks using abstraction and regular languages. In *CONCUR*, vol. 962 of *LNCS*, pp. 395–407. Springer, 1995.
4. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, vol. 2619 of *LNCS*, pp. 331–346. Springer, 2003.
5. A. Cohen and K. S. Namjoshi. Local proofs for linear-time properties of concurrent programs. In *CAV*, vol. 5123 of *LNCS*, pp. 149–161. Springer, 2008.
6. S. J. Creese and A. W. Roscoe. Data independent induction over structured networks. In *PDPTA*. CSREA Press, 2000.
7. E. A. Emerson and V. Kahlon. Parameterized model checking of ring-based message passing systems. In *CSL*, vol. 3210 of *LNCS*, pp. 325–339. Springer, 2004.
8. E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *CAV*, vol. 697 of *LNCS*, pp. 463–478. Springer, 1993.
9. N. Ghafari, A. Gurfinkel, and R. J. Trefler. Verification of parameterized systems with combinations of abstract domains. In *FMOODS/FORTE*, vol. 5522 of *LNCS*, pp. 57–72. Springer, 2009.
10. C. N. Ip and D. L. Dill. Verifying systems with replicated components in murphi. *Formal Methods in System Design*, 14(3):273–310, 1999.
11. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *CAV*, vol. 1254 of *LNCS*, pp. 424–435. Springer, 1997.
12. Y. Kesten, A. Pnueli, E. Shahar, and L. D. Zuck. Network invariants in action. In *CONCUR*, vol. 2421 of *LNCS*, pp. 101–115. Springer, 2002.
13. S. La Torre, P. Madhusudan, and G. Parlato. Analyzing recursive programs using a fixed-point calculus. In *PLDI*, pp. 211–222. ACM, 2009.
14. S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *CAV*, vol. 5643 of *LNCS*, pp. 477–492. Springer, 2009.
15. S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.
16. A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *CAV*, vol. 5123 of *LNCS*, pp. 37–51. Springer, 2008.
17. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pp. 446–455. ACM, 2007.
18. A. Pnueli, J. Xu, and L. D. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In vol. 2404 of *LNCS*, pp. 107–122. Springer, 2002.
19. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, vol. 3440 of *LNCS*, pp. 93–107. Springer, 2005.
20. S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI*, pp. 14–24. ACM, 2004.
21. D. Suwimonteerabuth, J. Esparza, and S. Schwoon. Symbolic context-bounded analysis of multithreaded java programs. In *SPIN*, vol. 5156 of *LNCS*, pp. 270–287. Springer, 2008.

22. T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent linux device drivers. In *ASE*, pp. 501–504. ACM, 2007.

Appendix

A Proof of Lemma 3

Lemma 3. *Let* $\overline{u} = (u_1, \ldots, u_k)$*,* $\overline{v} = (v_1, \ldots, v_k)$*,* σ *such that* (σ, v_i) *is a localized state of* $\mathcal{P}$*,* $k \in \mathbb{N}$*, and* $i \leq k$*.*

1. *TLI*$(i, \sigma, \overline{u}, \overline{v})$ *holds iff there is an execution* π *of* $\mathcal{P}$ *under a* k*-round schedule such that* $(\overline{u}_i, \overline{v}_i)$ *is a thread linear interface covered by* π *and* (σ, v_i) *is a final localized state of* $(\overline{u}_i, \overline{v}_i)$*.*
2. *RLI*$(i, \overline{u}, \overline{v})$ *holds iff there is an execution* π *of* $\mathcal{P}$ *under a* k*-round schedule such that* $(\overline{u}_i, \overline{v}_i)$ *is a right linear interface covered by* π*.*
3. *WRLI*$(i, \overline{u}, \overline{v})$ *holds iff there is an execution* π *of* $\mathcal{P}$ *under a* k*-round schedule such that* $T_1, \ldots, T_m$ *are scheduled at the end of each round,* u_i *is the shared state on context-switching to* T_1 *along* π *in round* i*,* $i > 1$*, and* $(\overline{u}_{i-1}, \overline{v}_{i-1})$ *is a right linear interface covered by* π *on* $T_1, \ldots, T_m$*.*

Proof. The "only if" direction of the lemma is proved by induction on the number of steps taken by the evaluation of the fixed-point equations defining TLI, RLI, and *WRLI*. In the first iteration, only predicate TLI holds true for all tuples $(1, \sigma_1, \overline{u}, \overline{v})$ such that *LocInit*(σ_1) holds, $u_1 = v_1$, and *ShInit*(u_1) holds. Thus, any k-round execution π of $\mathcal{P}$ starting from an initial state $(map, i, \nu, \sigma_1, \ldots, \sigma_m)$ in which all the threads do not do any move along π is such that $(\overline{u}_1, \overline{v}_1)$ is an initial thread linear interface of π whose final localized state is (σ_1, v_1).

The inductive step is case-based and here we only consider one case for *WRLI*. All the other cases can be shown similarly. Suppose that *WRLI*$(i, \overline{u}, \overline{v})$ is true due to the fact that *TLI*$(i, \sigma, \overline{w}, \overline{u})$, *RLI*$(i-1, \overline{u}, \overline{v})$, and *WRLI*$(i, \overline{w}, \overline{v})$ all hold true, for some σ and $\overline{w}$. By the inductive hypothesis, we consider (1) a run π' witnessed by *WRLI*$(i, \overline{w}, \overline{v})$ which can be decomposed into two linear interfaces $L = (\overline{z}_i, \overline{w}_i)$ which is initial and $R' = (\overline{w}_i, \overline{v}_i)$, (2) the thread linear interface $(\overline{w}_i, \overline{u}_i)$ whose existence is due to *TLI*$(i, \sigma, \overline{w}, \overline{u})$ that holds true, and (3) the right linear interface $R = (\overline{u}_{i-1}, \overline{v}_{i-1})$ which is guaranteed to exist by th! e fact that *RLI*$(i-1, \overline{u}, \overline{v})$ holds true. Let R'' be the pair $(\overline{u}_i, \overline{v}'_i)$ which extend R as follows: $\overline{u}_i$ is the same as $\overline{u}_{i-1}$ except that in the last position contains u_i, in similar way let $\overline{v}_i$ be the extension of $\overline{v}'_{i-1}$ having u_i as last component. Intuitively, R'' represent the same computation block of R except the in the last round all the threads covered by R do not do any move and hence propagate the shared state u_i along the computation on round i in R''. Since L, T and R'' compose in this order, the resulting linear interface B wraps and is also initial. Thus, by Lemma 2 applied to the linear interface B there is an execution π of $\mathcal{P}$. Notice that π' satisfies condition 3 of the lemma, by picking $T_1, \ldots T_m$ as a sequence of threads covered by R''.

Here we prove the "if" direction. The proof is by induction. For the first assertion we indeed prove a stronger property: if there is an execution π of $\mathcal{P}$ under a k-round schedule such that $(\overline{u}_i, \overline{v}_i)$ is a thread linear interface covered by π and (σ, v) is a final localized state of $(\overline{u}_i, \overline{v}'_i)$ where $\overline{v}'_{i-1} = \overline{v}_{i-1}$ and $v'_i = v$, then $TLI(i, \sigma, \overline{u}, \overline{v}')$ holds. In the following, we will refer to this property as assertion 1, and as assertion 2 and 3 respectively the "if" part of the properties stated respectively in part 2 and 3 of the lemma.

We fix a k-round execution π and let $T^1, \ldots, T^h$ be the thread schedule in each round of π.

In the induction, π is explored by increasing round indexes, and within each round, in two phases: first from T^1 through T^h (to show assertions 1 and 3), and then from T^h through T^1 (to show assertion 2).

The base case corresponds to the start of π. Let $(map, 1, \nu, \sigma_1, \ldots, \sigma_h)$ be the first state in π. Since clearly $LocInit(\sigma_1)$ and $ShInit(\nu)$ both hold, then for part (1.1) of the definition of TLI, we get that $TLI(1, \sigma_1, \overline{u}, \overline{u})$ also holds for $u_1 = \nu$.

The induction step has several cases: moving forwards in the execution along a internal transition to a process (1), or across a context switch within a round (2) or across rounds (3), or moving backwards within a round at the last scheduled thread (4) or at any other thread (5). We observe that, in the cases corresponding to the forward exploration of π, i.e., cases (1), (2) and (3), there are no new right blocks of π which are discovered, therefore to show the induction step for assertion 2 it is sufficient the induction hypothesis. Analogously, in the backward exploration of π, the induction step for assertions 1 and 2 is by induction hypothesis. Also, in case (1), there is no new thread linear interface of π which is discovered, therefore assertion 3 follows from the induction hypothesis. In the remaining cases, we reason as follows.

In case (1), for assertion 1, the induction step is ensured by part (1.3) of the definition of TLI and on the fact that this formula refers only to tuples of TLI which have been already computed by induction hypothesis.

Now, in the forward exploration of round i of π, consider the context switch from state $s = (map, j, \nu, \sigma_1, \ldots, \sigma_m)$ to state $s' = (map, j + 1, \nu, \sigma_1, \ldots, \sigma_m)$ (case (2)). Denote with (σ, ν) and (σ', ν) respectively the localized states at s and s', with $(\overline{u}, \overline{v})$ and $(\overline{v}, \overline{w})$ the thread linear interfaces covered by π respectively on T^j and T^{j+1}, and with $(\overline{v}, \overline{z})$ the right linear interface covered by π on $T^{j+1}, \ldots, T^h$. Clearly, $\overline{v}_i = \nu$ holds.

If $i = 1$, by induction hypothesis, $TLI(1, \sigma, \overline{u}, \overline{v})$ holds, and by part (1.1) of the definition of TLI, we get that $TLI(1, \sigma', \overline{v}, \overline{v}')$ hold where $v'_1 = v_1$ ($LocInit(\sigma')$ must hold since computation of thread T^{j+1} starts at s' in π). (Also note that for $i = 1$, the relation $WRLI$ is not defined.) If $i > 1$ instead, by induction hypothesis, $TLI(i{-}1, \sigma', \overline{v}, \overline{w})$ and $RLI(i{-}1, \overline{v}, \overline{z})$ hold, and if $j > 1$, $WRLI(i, \overline{u}, \overline{z})$ also holds. Observe that, if $j = 1$, $ShInit(u_1)$ must hold (initial shared state of π) and $z_r = u_{r+1}$ for $r \in [1, i-1]$. Therefore, from the definition of $WRLI$ also $WRLI(i, \overline{v}, \overline{z})$ must hold. Moreover, from $TLI(i-1, \sigma', \overline{v}, \overline{w})$, $RLI(i-1, \overline{v}, \overline{z})$, $WRLI(i, \overline{v}, \overline{z})$ and part (1.2) of the definition of TLI also $TLI(i, \overline{v}, \overline{v}')$ must hold, for $\overline{v}'$ such that $\overline{v}'_{i-1} = \overline{w}_{i-1}$ and $v'_i = v_i$.

In case (3), consider the context switch from state $s = (map, h, \nu, \sigma_1, \ldots, \sigma_m)$ in round $i-1$ to state $s' = (map, 1, \nu, \sigma_1, \ldots, \sigma_m)$ in round i. Denote with (σ, ν) the localized state at s', with $(\overline{u}, \overline{v})$ the thread linear interface covered by π on T^1 and with $(\overline{v}, \overline{z})$ the right linear interface covered by π on $T^2, \ldots, T^h$. Clearly, $\overline{v}_i = \nu$ holds. By induction hypothesis, $TLI(i-1, \sigma, \overline{u}, \overline{v})$ and $RLI(i-1, \overline{v}, \overline{z})$ hold. Thus, since $ShInit(u_1)$ must hold (initial shared state of π) and $z_r = u_{r+1}$ for $r \in [1, i-1]$, from part (1.2) of the definition of TLI also $TLI(i, \sigma, \overline{u}, \overline{v})$ holds. (Observe, that only assertion 1 needs to be shown in this case.)

Now we consider the cases (4) and (5). Let $(\overline{u}, \overline{v})$ be the thread linear interface covered by π on T^j and (σ, v_i) be a final localized state of $(\overline{u}_i, \overline{v}_i)$. If $j = h$, by induction hypothesis, $TLI(i, \sigma, \overline{u}, \overline{v})$ holds and if $i > 1$ then $WRLI(i, \overline{u}, \overline{v})$ also holds (recall that the backwards phase follows the forwards phase in each round). Therefore, from the definition of RLI, also $RLI(i, \overline{u}, \overline{v})$ must hold. If $j < h$ instead, denoting with $(\overline{v}, \overline{w})$ the right linear interface covered by π on $T^{j+1}, \ldots, T^h$, by induction hypothesis, $TLI(i, \sigma, \overline{u}, \overline{v})$ and $RLI(i, \overline{v}, \overline{w})$ both hold, and if $i > 1$ then $WRLI(i, \overline{v}, \overline{w})$ also holds. Therefore, from the definition of RLI, also $RLI(i, \overline{u}, \overline{w})$ must hold, and we are done with the "if" direction of the lemma. $\square$

B Proof of PSPACE-completeness

The fact that the reachability problem in our setting is in PSPACE is very simple and follows from our algorithm. We do not show it in the paper, as it makes sense only in the *explicit* representation of programs as finite-state machines, while our notation, exposition of the algorithm, the fixed-points, and the implementation all are geared to work on programs that define state-spaces implicitly.

Here is a sketch the proof of PSPACE-completeness. First, let us define the precise problem. We are given explicit state-machines as processes with transitions of the form $(s, l) \rightarrow (s', l')$ which means that from shared state s, the process in local state l can transform its local state to l' and transform the shared state to s'. The reachability problem is: given k (in unary), is the error state reachable in k rounds. The fixed-point algorithm we give implements a fixed-point of a relation over $L \times S^k$, and is hence implementable in PSPACE. PSPACE-hardness follows from a reduction of the membership problem for linear-bounded automata (TMs with linear space) to this problem. Intuitively, we can model the initial tape of the TM using the inputs $\overline{u}_i$ to the first thread, model one step of the TM using one thread that acts as a transducer to change the tape by transforming the output shared variables, and the local states of a single process help look at neighboring cells to effect the transformation correctly. The EXPSPACE-hardness of the model in [1], follows from the results in [1]. The main difference between the two models is that in our model, we need to pay a context-switch for every thread creation, while the model in [1] allows thread creation without this price. See the section on Related Work and the footnote on Page 4.