Thread Modularity at Many Levels

A Pearl in Compositional Verification

Jochen Hoenicke
Albert-Ludwigs-Universität
Freiburg, Germany
hoenicke@informatik.uni-freiburg.de

Rupak Majumdar
MPI SWS
Kaiserslautern, Germany
rupak@mpi-sws.org

Andreas Podelski
Albert-Ludwigs-Universität
Freiburg, Germany
podelski@informatik.uni-freiburg.de

Abstract

A thread-modular proof for the correctness of a concurrent program is based on an inductive and interference-free annotation of each thread. It is well-known that the corresponding proof system is not complete (unless one adds auxiliary variables). We describe a hierarchy of proof systems where each level k corresponds to a generalized notion of thread modularity (level 1 corresponds to the original notion). Each level is strictly more expressive than the previous. Further, each level precisely captures programs that can be proved using uniform Ashcroft invariants with k universal quantifiers. We demonstrate the usefulness of the hierarchy by giving a compositional proof of the MACH shootdown algorithm for TLB consistency. We show a proof at level 2 that shows the algorithm is correct for an arbitrary number of CPUs. However, there is no proof for the algorithm at level 1 which does not involve auxiliary state.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification

Keywords Thread-modularity, Owicki-Gries, Concurrency, Parameterized verification

1. Introduction

Verifying concurrent programs against safety properties is a fundamental problem in computer-aided verification. Bugs in concurrent programs can arise due to unanticipated schedules between threads. In practice, such bugs are difficult to test for, reproduce, or debug. A naïve approach to verification constructs a sequential program that runs all possible interleavings of the threads in the original program (the "product construction"). The safety property is verified on the product program by identifying an inductive invariant. The product construction approach does not scale because the number of potential schedules grows exponentially with the number of threads. More seriously, the product construction does not work in the *parameterized* setting, in which we want to verify a program no matter how many threads are running in parallel.

POPL'17, January 15–21, 2017, Paris, France
ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009893

The classical approach to verification of concurrent programs uses a different, modular, approach. In this approach, one reasons about a single thread of the original program, summarizing the effect of all other threads that run in parallel. To prove an invariant, one infers an inductive invariant Inv for the single thread which is additionally *non-interfering*: if the state of the tracked thread satisfies Inv and some *other* thread takes a step, non-interference guarantees that Inv continues to hold for the updated state. This approach, called *thread-modular reasoning* [28] in the world of model checking and the *Owicki–Gries proof system* [37, 40, 47] in the world of program logics, can prove safety properties of many parameterized multi-threaded programs. Indeed, thread-modular reasoning is the basis for many software verification tools [3, 10, 18–20, 22, 26–28, 30–32, 34, 36, 38, 39, 43, 51, 52].

It is well-known that thread-modular reasoning is incomplete [6]: a program may satisfy a safety property but there may not be a thread-modular proof (i.e., a non-interfering inductive invariant Inv) demonstrating it. The incompleteness is not a theoretical curiosity; many practical examples, such as the ticket protocol for mutual exclusion, do not have thread-modular proofs. A natural challenge is to design proof systems that are suitable for practical examples while retaining the nice modularity properties of thread-modular reasoning. Theoretically, thread-modular reasoning can be made complete by adding auxiliary state [46, 47]. However, in practice, it is often not clear how to systematically introduce enough auxiliary state to regain precision while keeping the state space small (although initial work exists on the synthesis of auxiliary counter variables [24]). Independently of the ease of adding auxiliary state, it is worthwhile to investigate whether we can increase the expressiveness of thread-modular reasoning without adding arbitrary auxiliary state.

In this paper, we investigate a hierarchy of proof systems, *k-thread modular* proofs, between thread modularity and full product construction. At each level $k \geq 1$, k-thread-modular reasoning generalizes thread-modular reasoning (for $k = 1$, we find the original version of thread-modular reasoning, or equivalently, the original Owicki-Gries proof system). The notion of a thread-modular proof at level k, for $k \geq 1$, is based on two concepts: the inductive annotation for the product of k threads by assertions, which is as in thread-modular reasoning, and the *non-interference* at level k, which expresses the precondition for the action of an additional $(k + 1)$-st thread by a conjunction of k assertions (each conjunct stems from an assertion in the inductive annotation for the product of the k threads). The resulting hierarchy is *strict*: for each $k > 1$, there exist programs, even ones where each thread is finite-state, which can be proved correct by a thread-modular proof at level k but not by any thread-modular proof at level $k - 1$. Further, the hierarchy is incomplete for infinite-state threads; thus, the addition of auxiliary state cannot be dispensed with in general.

Natural questions about the hierarchy of thread-modular proof systems are about soundness, the power of the proof system at level k, or detecting that no proof at level k exists.

We characterize the power of thread-modular proofs at level k by showing a correspondence between such proofs and *Ashcroft invariants* with k universal quantifiers. Ashcroft invariants [8] can prove concurrent programs over a fixed number of threads by using assertions over the global states, where *program counter* variables indexed by thread identifiers refer to the local control location of each thread. An Ashcroft invariant as in [25] uses universal quantifiers over thread identifiers to express assertions over the global state of a program with an unbounded number of threads.[1] We show a relative completeness result: any program provable using an Ashcroft invariant with k universal quantifiers admits a thread-modular proof at level k and vice versa. From the soundness of Ashcroft invariants, we derive in one shot soundness for thread-modular proofs at level k for each $k \geq 1$. A corollary of our relative completeness result is that thread-modular proofs (without auxiliary variables!) are complete for finite-state threads (see also [4, 32] for different formulations of the same result).

In order to detect that no thread-modular proof at level k exists, we encode the existence of an invariant for a thread-modular proof at level k through a *Horn clause constraint* as in [30, 32, 35]. We show that there is no thread-modular proof at level k if and only if there is a ground derivation of false for the Horn clause encoding. Further, the Horn clause encoding enables us to prove strictness of the hierarchy (see also [35]).

We show that higher levels of thread modularity are not just a theoretical curiosity by presenting a number of examples which cannot be proved using thread-modular reasoning at level 1 but which can be proved using thread-modular reasoning at level 2. In particular, we prove the correctness of a non-trivial algorithm: the TLB shootdown algorithm in the Mach operating system [13]. We show that the key safety invariant for this protocol can be proved for an arbitrary number of concurrent processors by a proof at $k = 2$ (but not at $k = 1$). An indication of the complexity of the proof is perhaps the fact that the existence of a subtle race condition in the original —rather old— algorithm had not been detected before. The race may be triggered in modern hypervisor systems [12]. We prove correctness of a slightly modified version that prevents the race.

In summary, we have the following facts about k-thread-modular proof systems.

1. Thread-modular proof systems at level k increase the expressive power of thread modularity without introducing arbitrary auxiliary state.
2. The power of the proof system at level k is exactly characterized by uniform Ashcroft invariants with k universal quantifiers.
3. The induced hierarchy is strict, already for finite-state threads. The hierarchy is incomplete: there are correct parameterized programs which cannot be proved by thread-modular proofs at any level k. Further, it is semi-decidable if a program does not have a proof at level k.
4. For finite-state threads, the k-thread-modular proofs are complete: for every program and safety property, if the program satisfies the property, then there is a k-thread-modular proof for some k.

The contribution of the results above does not lie in their novelty. Indeed, most of the results have been known before and have been used, implicitly or explicitly, in work on automatic verification for parameterized systems (see, e.g., [4, 5, 11, 14, 35, 45, 51, 53]). Instead, the contribution of our work lies in putting together these results in one uniform framework and showing that k-thread-modular reasoning forms a basis for verifying systems that may presently be out of the scope of existing approaches to automatic verification.

The remainder of the paper is organized as follows. Section 2 introduces a parameterized model of multi-threaded shared memory concurrency. Section 3 defines the thread-modular proof system at level k and Section 4 shows that the proof system forms a strict hierarchy for parameterized systems with increasing k. Section 5 connects thread-modular proofs at level k with Ashcroft invariants with k quantifiers. In Section 6, we augment our programming model with *synchronous statements*, which can affect the values of local variables of multiple threads. Synchronous statements are required to model multi-processor protocols; we give the example of Mach's TLB shootdown algorithm and its proof at level $k = 2$. We conclude with related work (Section 7) setting our work in context.

2. Preliminaries

We will define a program P through a thread template T. A thread template is a sequential program in which one distinguishes between *global* and *local variables*. The program consists of an unbounded number of threads where each thread runs the same thread template, or rather the instance of the thread template with its own copy of the local variables. Thus, a thread can read and write global variables and its local variables, but not the local variables of any other thread. In particular, the execution of a thread cannot depend on the existence of some other thread in a specific state. Having only a single thread template is not a proper restriction since multiple thread templates can always be combined into one (using global flags and a *big switch statement* in the beginning).

For the formal definition of a thread template, we fix a tuple g of variables that we will call *global*, and a tuple x of variables that we will call *local*. We will see that each instance of the thread template will get its own copy of local variables, and that we will use $x_1, \ldots, x_n$ to refer to the copies of tuples of local variables for n individual threads.

The set of statements Stmts for a thread template refers to the global and local variables in g resp. x. The syntax of statements $s \in \mathsf{Stmts}$ is defined by the grammar

$$s \ ::= \ [\varphi] \mid v := e \mid s_1; s_2$$

using assume statements, assignments, and sequential composition, where φ is a Boolean expression over global and local variables, e is an expression over global and local variables, and v is one of the global or local variables in g resp. x.

A *thread template* $T = (\mathsf{Loc}, \delta, \ell_{\mathsf{init}})$ is a control flow graph, i.e., a graph with a set Loc of nodes called program locations, and a set of labeled edges $\delta \subseteq \mathsf{Loc} \times \mathsf{Stmts} \times \mathsf{Loc}$. Thus, each edge $(\ell_{src}, s, \ell_{tgt})$ is labeled by a statement $s \in \mathsf{Stmts}$. The initial location is $\ell_{\mathsf{init}} \in \mathsf{Loc}$.

Intuitively, in each execution step of the program, one of the threads moves its control along an edge $(\ell_{src}, s, \ell_{tgt})$ in the graph and executes the statement s labeling the edge atomically (only one thread moves at a time).

We use the *thread identifiers* $1, \ldots, n$ to distinguish n threads. For $i = 1, \ldots, n$, we use T_i to refer to the instance of the thread template T for the thread i. The instance T_i is obtained by taking the control flow graph for T and replacing each statement s by the statement $(s : i)$. Thus, $(\ell_{src}, (s : i), \ell_{tgt})$ is an edge in T_i if $(\ell_{src}, s, \ell_{tgt})$ is an edge in T. The statement $(s : i)$ is obtained by replacing each local variable by its i-th copy. We will use x_i for

[1] Ashcroft's original paper only considered concurrent programs with a fixed number of threads. Recently, [25] extended the method to parameterized programs by introducing universal quantification over thread ids.

the tuple of local variables of thread i (i.e., for the i-th copy of the tuple of local variables x).

We define $P(n)$, the instance of the parameterized program P for the natural number n, as the parallel composition of $T_1, \dots, T_n$.

$$P(n) = T_1 \,||\, \dots \,||\, T_n$$

Avoiding a general definition of parallel composition, we represent the program $P(n)$ by its control flow graph, which is defined as the *product* of the control flow graphs for $T_1, \dots, T_n$. The set of locations of $P(n)$ is Loc^n. Thus, a location of $P(n)$ is of the form $(\ell_1, \dots, \ell_n)$ where $\ell_1, \dots, \ell_n$ are locations of the thread template T. Each edge in $P(n)$ is labeled by a statement of the form $(s : i)$ for some i between 1 and n and corresponds to an edge labeled s in the control flow graph for T. Formally,

$$\begin{aligned}((\ell_1, \dots, \ell_{i-1}, \ell_{src}, \ell_{i+1}, \dots, \ell_n),\\ (s : i), (\ell_1, \dots, \ell_{i-1}, \ell_{tgt}, \ell_{i+1}, \dots, \ell_n))\end{aligned}$$

is an edge in $P(n)$ whenever $(\ell_{src}, s, \ell_{tgt})$ is an edge in the thread template T.

The control flow graph for the program $P(n)$ defines a sequential program (sometimes called the *interleaving semantics* or the *non-deterministic sequentialization* of the parallel composition of $T_1, \dots, T_n$). We can thus apply the same concepts as for sequential programs in order to define notions such as: state, execution, correctness, the validity of Hoare triples, etc. for $P(n)$.

The variables of $P(n)$ are given by g, x_1, ..., x_n, i.e., the tuple of global variables and the n tuples of (the copy of) the local variables of each thread.

A state of $P(n)$ is a pair $\sigma = ((\ell_1, \dots, \ell_n), \nu)$ of a program location of $P(n)$ and a valuation of the variables of $P(n)$. A state σ of $P(n)$ thus specifies the values of the global variables ("$\nu(g)$") and, for each of the n threads, the control location ("ℓ_i") and the values of the corresponding copy of local variables of the thread ("$\nu(x_i)$").

The notion of correctness of P will refer to initial states and to error states, notions which we introduce next. The definitions of initial state resp. error state must be *uniform*, i.e., applicable to $P(n)$ for every $n \geq 1$.

We use the symbol init to refer to a set of initial valuations for P, i.e., a union of sets of initial valuations of $P(n)$ for every $n \geq 1$. We assume that init is defined by a condition on the valuation of global variables and the local variables of each thread; i.e., init is of the form $\mathsf{init} = \{\nu \mid \nu(g) \in \mathsf{init}_{\mathsf{global}} \wedge \forall i. \nu(x_i) \in \mathsf{init}_{\mathsf{local}}\}$ for some sets $\mathsf{init}_{\mathsf{global}}$ and $\mathsf{init}_{\mathsf{local}}$. An initial state is of the form $\sigma = ((\ell_1, \dots, \ell_n), \nu)$ where each of $\ell_1, \dots \ell_n$ is the initial location ℓ_{init} of the thread template T and ν is an initial location, i.e., $\nu \in \mathsf{init}$. An execution of $P(n)$ must start in an initial state.

We use the symbol err_n to refer to the set of error states of $P(n)$. We assume that err_n is defined by a *generator set* err_m of dimension m, i.e.,

$$\begin{aligned}\mathsf{err}_n = \{((\ell_1, \dots, \ell_n), \nu) \mid ((\ell_{i_1}, \dots, \ell_{i_m}), \nu') \in \mathsf{err}_m.\\ \nu'(g) = \nu(g), \nu'(x_j) = \nu(x_{i_j}) \text{ for } j \in \{1, \dots, m\}\\ \text{for some } \nu', i_1, \dots, i_m \text{ s.t. } 1 \leq i_1 < \dots < i_m \leq n\}\end{aligned}$$

for some set of states err_m of $P(m)$. Thus, for $n < m$, err_n does not contain any states; in other words, $P(n)$ does not have any error states. For example, to capture *thread reachability*, err can be defined by a generator set of dimension 1. Then, err_n is the set of locations $((\ell_1, \dots, \ell_n), \nu)$ such that for one $i \in \{1, \dots, n\}$ and ν' with $\nu'(g) = \nu(g), \nu'(x_1) = \nu(x_i)$, the state $((\ell_i), \nu')$ is in err_m. To specify *mutual exclusion*, err can be defined by a generator set of dimension 2, say, $\mathsf{err}_m = \{((\ell_{\mathsf{crit}}, \ell_{\mathsf{crit}}), \nu) | \nu \text{ valuation}\}$ where ℓ_{crit} is a distinguished *critical* location of the thread template T. Then, err is the set of locations $(\ell_1, \dots, \ell_n)$ such that two out of $\ell_1, \dots \ell_n$ form a pair of *critical* locations (i.e., for some $i < j$, $\ell_i = \ell_{\mathsf{crit}}$ and $\ell_j = \ell_{\mathsf{crit}}$).

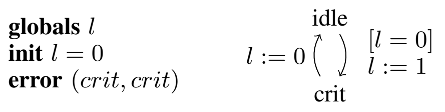

Figure 1. Mutual exclusion with a lock

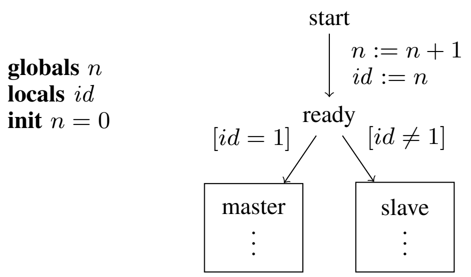

Figure 2. Encoding thread identifiers

From now on, we assume that the thread template T, the set of initial states init, and the set of error states err_m is fixed. We define $P(n)$ to be correct if an error state is not reached in any execution of $P(n)$. If $P(n)$ is correct, then $P(m)$ is correct for every $m \leq n$ (this is simply because every execution of m threads corresponds to an execution of n threads where $n - m$ do not move). We define that the program P is correct if $P(n)$ is correct for every $n \geq 1$.

Example 1. ***[Mutual exclusion with a lock [28]]*** Figure 1 shows a thread template for a simple mutual exclusion protocol using a global lock l. The template has two locations, idle and crit, and the initial location is idle. The initial valuation sets the global variable l to 0. The edge from idle to crit performs an atomic test-and-set: if l is 0, it is set to 1. The edge from crit back to idle sets l back to 0. A program with arbitrarily many copies of this template satisfies mutual exclusion: no two threads are simultaneously at crit. We can specify the error condition using a generator set of dimension 2 of all states of the form $((\text{crit}, \text{crit}), \nu)$.

Example 2. ***[Encoding thread identifiers]*** While we define programs using parallel composition of a single thread template, we can encode several common programming idioms. Figure 2 shows an example template which assigns a unique thread identifier to each thread and ensures that the thread with identifier 1 (the "master") executes code different from all other threads (the "slaves"). The global variable n is used as a source of unique identifiers; each thread initially assigns itself a unique identifier by incrementing n and copying the current value atomically to its local variable id. We use this initialization trick in the TLB shootdown example in Section 6. Moreover, in the example, the thread with identifier 1 executes code different from the other threads. The idea can be generalized to encode any finite number of distinct templates.

3. Thread Modularity at Level k

We now introduce the proof system and illustrate it with some examples.

3.1 The Proof System

An *assertion* φ for $P(n)$ is a formula over the variables of $P(n)$. It denotes a set of valuations ν of global variables g and local variables $x_1, \dots, x_n$ for each of the n threads (namely, the set of all valuations ν that satisfy φ). An assertion for $P(n)$ can be evaluated on a state of $P(m)$ for every $m \geq n$. We shall assume that the assertion language contains the logical system of *elementary arithmetic* [49], the language of the natural numbers with addition and multiplication.

Given some $k \geq 1$, we will define a proof rule that uses assertions φ over states of $P(k)$ in order to prove the correctness of $P(n)$ for every $n \geq k$ (and thus also for every $n \geq 1$). The proof rule will use these assertions φ also over states of $P(k+1)$.

We transform φ into the formula $\varphi[x_{k+1}/x_1]$ by substituting x_{k+1} for x_1, which is a formula over the local variables $x_{k+1}, x_2, \ldots, x_k$ (of thread $k+1$, thread 2, ..., thread k). Similarly for $\varphi[x_{k+1}/x_2]$, which is a formula over the local variables $x_1, x_{k+1}, x_3, \ldots, x_k$ (of thread 1, thread 3, ..., thread k, and thread $k+1$), etc. Just as we use φ, a formula over the (tuples of) variables $x_1, \ldots, x_k$, as an assertion over states of $P(k+1)$, we can use the formula $\varphi[x_{k+1}/x_i]$ (obtained by substituting x_{k+1} for x_i) as an assertion over states of $P(k+1)$. This is what we will do below.

A Hoare triple for $P(n)$ is formed by assertions φ and ψ for $P(n)$ and the instance of a statement s by thread i, for some i between 1 and n.

$$\{\varphi\}\ (s : i)\ \{\psi\}$$

The Hoare triple above is valid for $P(n)$ if the following holds for every pair of states σ and σ' of $P(n)$: if σ satisfies φ and σ can go to σ' under the execution of the statement $(s : i)$ by thread i, then σ' satisfies ψ.

An *annotation* Φ for $P(n)$ is a map from Loc^n to assertions φ for $P(n)$. Thus,

$$\Phi(\ell_1, \ldots, \ell_n)$$

is an assertion for $P(n)$ (i.e., a formula φ over the global variables g and the local variables x_1 of thread 1, ..., the local variables x_n of thread n).

We assume that the set of initial valuations is defined by an assertion. From now on, init refers to the assertion over global variables g and local variables x that defines the set of initial valuations for the thread template. Then $\bigwedge_{i=1}^{n} \mathsf{init}[x_i/x]$ defines the initial valuations of $P(n)$. Likewise, $\mathsf{err}(\ell_1, \ldots, \ell_n)$ refers to the assertions that define the set of valuations ν with $((\ell_1, \ldots, \ell_n), \nu) \in \mathsf{err}_n$. It can be defined from the assertion for the generator set of dimension m as:

$$\bigwedge_{1 \leq i_1 < \ldots < i_m \leq n} \mathsf{err}_m(\ell_{i_1}, \ldots, \ell_{i_m})[x_{i_1}/x_1] \ldots [x_{i_m}/x_m]\,.$$

Definition 1. An annotation Φ for $P(k)$ is a *thread-modular proof* at level k if it is initial, inductive, interference-free, and safe.

(initial) The assertion $\varphi_0 = \Phi(\ell_{init}, \ldots, \ell_{init})$ at the initial location of $P(k)$ is entailed by the assertion init,

$$\mathsf{init}[x_1/x] \wedge \cdots \wedge \mathsf{init}[x_k/x] \models \varphi_0$$

(inductive) The assertions φ and ψ at the source resp. target location of an edge in $P(k)$ and the statement labeling the edge form a valid Hoare triple. That is, for the assertion $\varphi = \Phi(\ell_1, \ldots, \ell_{i-1}, \ell_{src}, \ell_{i+1}, \ldots, \ell_k)$ and the assertion $\psi = \Phi(\ell_1, \ldots, \ell_{i-1}, \ell_{tgt}, \ell_{i+1}, \ldots, \ell_k)$,

$$\{\varphi\}\ (s : i)\ \{\psi\}$$

is valid for $P(k)$ for each labeled edge $(\ell_{src}, s, \ell_{tgt})$.

(non-interference) Every assertion φ labeling a location in $P(k)$ is interference-free under the execution of thread $(k+1)$. That is, for the assertion $\varphi = \Phi(\ell_1, \ldots, \ell_k)$ and the assertions $\psi_1 = \Phi(\ell_{src}, \ell_2, \ldots, \ell_k), \ldots, \psi_k = \Phi(\ell_1, \ldots, \ell_{k-1}, \ell_{src})$,

$$\{\varphi \wedge \psi_1[x_{k+1}/x_1] \wedge \ldots \wedge \psi_k[x_{k+1}/x_k]\}\ (s : k+1)\ \{\varphi\}$$

is valid for $P(k+1)$ for each labeled edge $(\ell_{src}, s, \ell_{tgt})$.

(safe) For $k \geq m$ (m is the dimension of the generator set of error states), we require that for every location $(\ell_1, \ldots, \ell_k)$, the corresponding annotation forbids the first m threads to enter an error state, i. e.,

$$\mathsf{err}(\ell_1, \ldots, \ell_m) \wedge \Phi(\ell_1, \ldots, \ell_k) \models \mathit{false}$$

For $k < m$, we require for every m-tuple $(\ell_1, \ldots, \ell_m)$,

$$\mathsf{err}(\ell_1, \ldots, \ell_m) \wedge \\ \bigwedge_{1 \leq i_1 < \cdots < i_k \leq m} \Phi(\ell_{i_1}, \ldots, \ell_{i_k})[x_{i_1}/x_1] \ldots [x_{i_k}/x_k] \\ \models \mathit{false}\,.$$

Intuitively, it is not clear how this definition can be used to show the correctness of P for any number of threads. A thread-modular proof at level k is an annotation of $P(k)$ for only a single $k \geq 1$ and it only includes some extended properties for $P(k+1)$. Nonetheless, one can show that these conditions are already sufficient for the correctness of $P(n)$ *for any* $n \geq 1$. The following theorem will be a consequence of Lemma 3 in Section 5.1.

Theorem 1 (Soundness). *If, for some $k \geq 1$, the program P has a thread-modular proof at level k, then P is correct.*

3.2 Examples

Example 3. ***[Mutual exclusion, continued]*** For our example program (Example 1 in Section 2), a thread-modular proof at level 2 exists; take the annotation Φ below.

$$\Phi : \begin{array}{rl} (\mathit{idle}, \mathit{idle}) \mapsto & \mathit{true} \\ (\mathit{idle}, \mathit{crit}) \mapsto & l = 1 \\ (\mathit{crit}, \mathit{idle}) \mapsto & l = 1 \\ (\mathit{crit}, \mathit{crit}) \mapsto & \mathit{false} \end{array}$$

The program is used in [28] as a prototypical example for which no thread-modular proof (in our terminology: no thread-modular proof at level 1) exists. To see why this is the case, we can use Lemma 1, Part 2; see Section 4.

We next discuss two classical examples for parameterized programs. Note that these examples do not have thread-modular proofs at level 1, again by Lemma 1 below.

Example 4. ***[Ticket mutual exclusion protocol]*** Figure 3(a) shows a template for the ticket mutual exclusion protocol. The protocol maintains two global integers, s and t, initially equal. Each thread has a local copy of the variable m. In order to go to the critical section (location crit), a thread first atomically copies the current value of the "ticket" t into its local variable m and increments t and

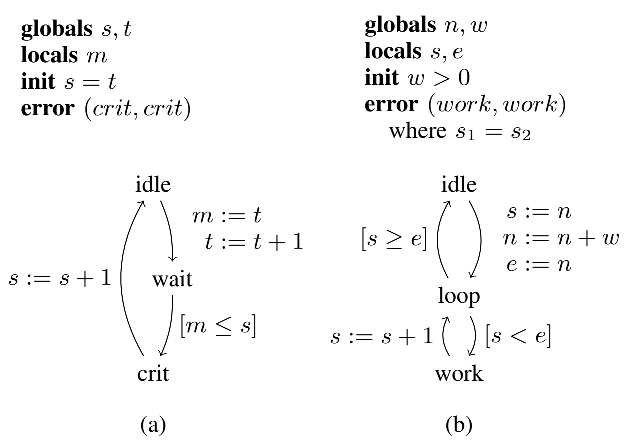

Figure 3. (a) Ticket protocol, (b) Thread pooling

second, waits until its local value is less than or equal to s. To move out of the critical section, a thread increments s. There is an error if more than one thread is in the critical section at the same time. Again, there is no proof with $k = 1$. There is a proof with $k = 2$, as illustrated in Figure 4.

$$\Phi : \begin{array}{rcl} (idle, idle) & \mapsto & s \leq t \\ (wait, idle) & \mapsto & s \leq m_1 < t \\ (wait, wait) & \mapsto & (\quad s \leq m_1 < m_2 < t \\ & & \lor\, s \leq m_2 < m_1 < t) \\ (crit, idle) & \mapsto & s < t \\ (crit, wait) & \mapsto & s < m_2 < t \\ (crit, crit) & \mapsto & \mathit{false} \\ \vdots & & \text{(symmetric cases)} \end{array}$$

Figure 4. An annotation for ticket which forms a thread-modular proof at level 2. We omit the annotations for the symmetric cases, such as $(idle, wait)$, etc.

Example 5. ***[Thread pooling]*** Figure 3(b) shows a thread pooling example adapted from [51]. Each thread atomically reserves a list of work items in the edge from $idle$ to $loop$. The global variable n gives the number of the next unreserved work item and the variable w give the number of items that are reserved in the step. After this step the items from s to e are worked on without any synchronization with the other threads. The correctness criterion is that no two threads work on the same item $s_1 = s_2$. In this example, a set of threads co-operatively execute tasks from some global pool of tasks. A global variable n tracks the current task. Each thread picks up w tasks starting from the current task and executes the next w tasks sequentially (using local variables s(tart) and e(nd) to track their current position). In this example, two threads should not work on the same task. Thus, it is an error if there are two threads in the $work$ state whose local variables s are equal. Again, there is no thread-modular proof at level 1. Figure 5 shows an annotation for a thread modular proof at level 2.

$$\Phi : \begin{array}{rl} (idle, idle) & \mapsto w > 0 \\ (loop, idle) & \mapsto w > 0 \land s_1 \leq e_1 \leq n \\ (work, idle) & \mapsto w > 0 \land s_1 < e_1 \leq n \\ (loop, loop) & \mapsto w > 0 \land (\quad s_1 \leq e_1 \leq s_2 \leq e_2 \leq n \\ & \qquad\qquad \lor\, s_2 \leq e_2 \leq s_1 \leq e_1 \leq n) \\ (work, loop) & \mapsto w > 0 \land (\quad s_1 < e_1 \leq s_2 \leq e_2 \leq n \\ & \qquad\qquad \lor\, s_2 \leq e_2 \leq s_1 < e_1 \leq n) \\ (work, work) & \mapsto w > 0 \land (\quad s_1 < e_1 \leq s_2 < e_2 \leq n \\ & \qquad\qquad \lor\, s_2 < e_2 \leq s_1 < e_1 \leq n) \\ \vdots & \quad \text{(symmetric cases)} \end{array}$$

Figure 5. An annotation for thread pooling which forms a thread-modular proof at level 2. We omit the annotations for the symmetric cases, such as $(idle, loop)$, etc.

4. Strictness of the Hierarchy

As described in the introduction, we use a Horn clause encoding as in [30, 32, 35] in order to detect that no thread-modular proof at level k exists. There is no thread-modular proof at level k if and only if there is a ground derivation of false for the Horn clause encoding. Furthermore, the Horn clause encoding enables us to prove strictness of the hierarchy.

We next introduce the Horn clause encoding. Suppose that we are given template T and a level k. First, for each labeled edge $(\ell_{src}, s, \ell_{tgt})$ and thread identifier i, we perform a logical encoding of the statement s as a constraint between two states in the standard way. We write $s(g, x, g', x')$ for the logical constraint corresponding to statement s. For example, in the mutual exclusion example, the statement $[l = 0]; l := 1$ is encoded as the constraint $l = 0 \land l' = 1$.

Figure 6 shows a set of Horn constraints such that any solution Inv provides an annotation satisfying the conditions of Definition 1. The constraints use an (unknown) relation Inv with variables $g, \ell_1, \ldots, \ell_k$, and $x_1, \ldots, x_k$. The constraints correspond to the requirements on the annotation Φ in Definition 1. (We only present the case $k \geq m$ for "safe", the other case is analogous.)

We want to relate annotations $\Phi(\ell_1, \ldots, \ell_k)$ in a thread-modular proof with solutions to the Horn constraints. However, the annotations are in the language of assertions but the solutions to the Horn constraints are sets of states. Their correspondence depends on a well known result from recursion theory: any recursively enumerable set is definable in the language of elementary arithmetic [49] (note that the solutions to the Horn constraints are recursively enumerable sets). The translation between the solutions and the assertions is not effective, but for the completeness result, it suffices that the annotations always exist. In the following, with abuse of notation, we do not explicitly talk about the encoding of sets of solutions into assertions.

Lemma 1. *Assume an encoding of thread modularity at level k as a Horn clause constraint as given in Figure 6.*

1. *Let Inv be a solution to the Horn clause constraints. Define the annotation Φ as*
$$\Phi(\ell_1, \ldots, \ell_k) = Inv(g, \ell_1, x_1, \ldots, \ell_k, x_k).$$
Then, Φ is a thread-modular proof at level k.
2. *If there exists a ground Horn clause derivation of false, then there is no thread-modular proof at level k.*

Lemma 1 indicates a method to prove the absence of a thread-modular proof at level k. Theoretically, the corresponding problem is undecidable (but semi-decidable).

Example 6. ***[Mutual exclusion, continued]*** *Figure 7 shows the Horn clause encoding for thread-modular proofs at levels $k = 1$ and $k = 2$, respectively. For $k = 1$, there is a ground Horn clause derivation of false. By Lemma 1 (Part 2), there is no thread-modular proof of the program at level 1. For the existence of a thread-modular proof at level 2, see Example 3.*

We next state the strictness of the hierarchy.

Theorem 2. *For each $k > 0$, there is a parameterized program that has a thread-modular proof at level $k + 1$ but no thread-modular proof at level k.*

Proof. The example in Figure 8 can be viewed as a generalization of the program of Figure 1 from 1 to k (at most k threads can be in the criticial section). Instead of the global variable l (a *lock* with values 0 and 1), we now have the global variable *ctr* (with values 0, 1, 2, ...). Intuitively, it counts the number of threads that have entered a critical section. We consider the safety property is that the counter is never zero while at least one process is in the critical section. (As it will turn out, the proof of k-mutual exclusion will be part of the proof of this property.) For this example, there is a thread-modular proof of level $k + 1$. The proof uses the annotation Φ of $P(k + 1)$ where the assertion labeling a location states that *ctr* is at least the number of threads that are in *crit* in this location, but at most k, formally $\Phi(\ell_1, \ldots, \ell_k, \ell_{k+1}) = (n_0 \leq ctr \leq k)$

$$
\begin{array}{lll}
\text{(initial)} & \mathsf{init}(g, x_1) \wedge \cdots \wedge \mathsf{init}(g, x_n) \rightarrow \mathit{Inv}(g, \ell_{\mathsf{init}}, x_1, \ldots, \ell_{\mathsf{init}}, x_k) & \\
\text{(inductive)} & \mathit{Inv}(g, \ell_1, x_1, \ldots, \ell_i, x_i, \ldots, \ell_k, x_k) \wedge s(g, x_i, g', x_i') \rightarrow \mathit{Inv}(g', \ell_1, x_1, \ldots, \ell_i', x_i', \ldots, \ell_k, x_k) & \text{for } (\ell_i, s, \ell_i') \\
\text{(non-interference)} & \mathit{Inv}(g, \ell_1, x_1, \ldots, \ell_k, x_k) \wedge & \\
 & \mathit{Inv}(g, \ell^\dagger, x^\dagger, \ell_2, x_2, \ldots, \ell_k, x_k) \wedge & \\
 & \vdots & \\
 & \mathit{Inv}(g, \ell_1, x_1, \ldots, \ell_{k-1}, x_{k-1}, \ell^\dagger, x^\dagger) \wedge s(g, x^\dagger, g', \cdot) \rightarrow \mathit{Inv}(g', \ell_1, x_1, \ldots, \ell_k, x_k) & \text{for } (\ell^\dagger, s, \ \cdot\ ) \\
\text{(safe)} & \mathit{Inv}(g, \ell_1, x_1, \ldots, \ell_k, x_k) \wedge \mathsf{err}(g, \ell_1, x_1, \ldots, \ell_m, x_m) \rightarrow \mathit{false} & \text{for } k \geq m
\end{array}
$$

Figure 6. Horn clause encoding for thread modularity at level k (where (ℓ_i, s, ℓ_i') and $(\ell^\dagger, s, \ \cdot\ )$ refer to statement s on an edge leading from ℓ_i to ℓ_i' and, respectively, from $\ell^\dagger$ to some other location in the control flow graph)

$$
\begin{array}{l}
k = 1: \\
\quad l = 0 \rightarrow \mathit{Inv}(l, \mathit{idle}) \\
\quad \mathit{Inv}(l, \mathit{idle}) \wedge l = 0 \wedge l' = 1 \rightarrow \mathit{Inv}(l', \mathit{crit}) \\
\quad \mathit{Inv}(l, \mathit{crit}) \wedge l' = 0 \rightarrow \mathit{Inv}(l', \mathit{idle}) \\
\quad \mathit{Inv}(l, \ell_1) \wedge \\
\quad \mathit{Inv}(l, \mathit{idle}) \wedge l = 0 \wedge l' = 1 \rightarrow \mathit{Inv}(l', \ell_1) \\
\quad \mathit{Inv}(l, \ell_1) \wedge \\
\quad \mathit{Inv}(l, \mathit{crit}) \wedge l' = 0 \rightarrow \mathit{Inv}(l', \ell_1) \\
\quad \mathit{Inv}(l, \mathit{crit}) \wedge \mathit{Inv}(l, \mathit{crit}) \rightarrow \mathit{false} \\
k = 2: \\
\quad l = 0 \rightarrow \mathit{Inv}(l, \mathit{idle}, \mathit{idle}) \\
\quad \mathit{Inv}(l, \ell_1, \ell_2) \rightarrow \mathit{Inv}(l, \ell_2, \ell_1) \\
\quad \mathit{Inv}(l, \mathit{idle}, \ell_2) \wedge l = 0 \wedge l' = 1 \rightarrow \mathit{Inv}(l', \mathit{crit}, \ell_2) \\
\quad \mathit{Inv}(l, \mathit{crit}, \ell_2) \wedge l' = 0 \rightarrow \mathit{Inv}(l', \mathit{idle}, \ell_2) \\
\quad \mathit{Inv}(l, \ell_1, \ell_2) \wedge \\
\quad \mathit{Inv}(l, \mathit{idle}, \ell_2) \wedge \\
\quad \mathit{Inv}(l, \ell_1, \mathit{idle}) \wedge l = 0 \wedge l' = 1 \rightarrow \mathit{Inv}(l', \ell_1, \ell_2) \\
\quad \mathit{Inv}(l, \ell_1, \ell_2) \wedge \\
\quad \mathit{Inv}(l, \mathit{crit}, \ell_2) \wedge \\
\quad \mathit{Inv}(l, \ell_1, \mathit{crit}) \wedge l' = 0 \rightarrow \mathit{Inv}(l', \ell_1, \ell_2) \\
\quad \mathit{Inv}(l, \mathit{crit}, \mathit{crit}) \rightarrow \mathit{false}
\end{array}
$$

Figure 7. Horn clauses for mutual exclusion with locks

if n_0 is the number of ℓ_i's such that $\ell_i = \mathit{crit}$). This implies that $1 \leq \mathit{ctr}$ if $n_0 = 1$, i.e., if a thread is in the critical section then the value of ctr is different from 0 (no error state is reachable). This also implies that, if $k + 1$ threads are in crit in the location $(\mathit{crit}, \ldots, \mathit{crit}, \mathit{crit})$, the assertion labeling the location is *false* (which is entailed by $k + 1 \leq \mathit{ctr} \leq k$, taking $n_0 = k + 1$) which entails k-mutual exclusion.

There is, however, no thread-modular proof of level k. To see this, we will use Lemma 1 (part 2) and construct a ground Horn clause derivation of *false* in the Horn clause constraint that corresponds to level k. In this derivation, all k threads enter crit, i.e., $\mathit{Inv}(k, \mathit{crit}, \ldots, \mathit{crit})$. Using the non-interference clause for the $\mathit{ctr} := \mathit{ctr} - 1$ edge one can derive $\mathit{Inv}(k - 1, \mathit{crit}, \ldots, \mathit{crit})$. From this state one can reach the error condition by letting $k - 1$ threads leave the critical section. □

globals ctr
init $\mathit{ctr} = 0$
error crit
where $\mathit{ctr} = 0$

$$\begin{tikzcd}
\text{idle} \arrow[d, bend left, "{[\mathit{ctr} < k] \; \mathit{ctr} := \mathit{ctr} + 1}"] \\
\text{crit} \arrow[u, bend left, "{\mathit{ctr} := \mathit{ctr} - 1}"]
\end{tikzcd}$$

Figure 8. k-mutual exclusion

Theorem 3. *There exist correct parameterized programs that cannot be proved by thread-modular proofs at any level k.*

Proof. If we remove the guard $\mathit{ctr} < k$ in the program of Figure 8, the resulting program is still safe but there is no thread-modular proof for any k. This can be seen by applying the same reasoning using Lemma 1 (part 2) as above. This shows that the thread-modular proof method cannot be used to prove *every* correct program without adding auxiliary state. □

Theorem 2 continues to hold for the restriction of parameterized programs to a *finite-state* thread template (the program of Figure 8 used in the proof of Theorem 2 can be made finite-state) but Theorem 3 does not. That is, for every correct parameterized program over a finite-state thread template, a thread-modular proof does exist at some level k in the hierarchy (we will show this in Section 5; see Theorem 5). This means that no auxiliary variables are needed for the thread-modular proof of a parameterized program over a finite-state thread template. However, the completeness of thread modularity in the finite-state case comes at a price; the level k can become very high in comparison to the size of the template.

5. The Power of Thread Modularity at Level k

In this section, we will first introduce a variation of Ashcroft invariants and an extension of Cartesian abstraction. We will then be able to state the relative completeness of thread modularity at level k (Theorem 4). Sections 5.1 and 5.2 are devoted to the main lemmas of the proof. In Section 5.3 we will investigate the consequence of relative completeness to the case of threads with finite state.

Ashcroft Invariants. We will introduce the notion of an Ashcroft invariant for $P(n)$ (a concurrent program with n threads) through the notion of an invariant for the corresponding array program $P[n]$ (a sequential non-deterministic program with arrays of size n). Intuitively, we go from $P(n)$ to $P[n]$ as follows. We view the value of the local variable x in the thread t as the element $x[t]$ of an array x at position t. We *flatten* the control flow graph of $P(n)$ into one with one single node and many self-loops, namely one for every edge of $P(n)$. We accommodate the control flow of $P(n)$ by treating program locations as data in $P[n]$.

Formally, the local state of the array program $P[n]$ is given by valuations that assign to each local variable in X an array with the index set $Tid = \{1, \dots, n\}$. For a variable $x \in X$ and a thread $t \in Tid$, we write $x[t]$ for the value of the local variable x in the thread t. Furthermore, we introduce an array pc with the index set Tid that assigns to each thread the current location. The valuation for global variables does not change. The array program $P[n]$ consists of a single while loop which executes non-deterministically one of the transitions that we obtain as follows. We have a transition to $P[n]$ with the statement $[pc[t] = \ell_{src}]; s[t]; pc[t] := \ell_{tgt}$ for every edge $(\ell_{src}, s, \ell_{tgt}) \in \delta$ of the control flow graph of P and every thread $t \in Tid$. Here, $s[t]$ is the statement s where every occurrence of $x \in X$ is replaced by $x[t]$. The assume statement $[pc[t] = \ell_{src}]$ and the update $pc[t] := \ell_{tgt}$ accommodate the control flow (the thread t goes from the source to the target location of the edge).

The initial states are the states where $pc[t] = \ell_{\mathsf{init}}$ and $\mathsf{init}[x[t]/x]$ hold for every $t \in Tid$. The bad states are the valuations that satisfy

$$\begin{aligned}&pc[1] = \ell_1 \wedge \ldots \wedge pc[n] = \ell_n \wedge \\ &\mathsf{err}(\ell_1, \dots, \ell_n)[x[1]/x_1] \ldots [x[n]/x_n].\end{aligned}$$

Obviously, the program $P(n)$ is bisimilar to the array program $P[n]$. The bisimulation relation relates a state $((\ell_1, \dots, \ell_n), \nu)$ with the valuation $\nu^{\square}$ with $\nu^{\square}(g) = \nu(g)$, $\nu^{\square}(pc) = \{i \mapsto \ell_i | 1 \leq i \leq n\}$ and $\nu^{\square}(x) = \{i \mapsto \nu(x_i) | 1 \leq i \leq n\}$. It follows that $P(n)$ is safe if and only if $P[n]$ is safe.

An *array invariant* is defined to be a formula Ψ that satisfies the conditions below ($Tid = \{1, \dots, n\}$ is the set of array indices). The existence of an array invariant proves the correctness of the array program $P[n]$.

(initialization)

$$\bigwedge_{t \in Tid} (\mathsf{init}[x[t]/x] \wedge pc[t] = \ell_{init}) \models \Psi\,.$$

(continuation) For every $t \in Tid$ and every edge $(\ell_{src}, s, \ell_{tgt})$:

$$\{\Psi\}\ [pc[t] = \ell_{src}]; s[t]; pc[t] := \ell_{tgt}\ \{\Psi\}\,.$$

(safe) For every location tuple $(\ell_1, \dots, \ell_n)$:

$$\begin{aligned}&pc[1] = \ell_1 \wedge \cdots \wedge pc[n] = \ell_n \wedge \\ &\mathsf{err}(\ell_1, \dots, \ell_n)[x[1]/x_1] \ldots [x[n]/x_n] \wedge \Psi \quad \models \mathit{false}\,.\end{aligned}$$

Fix $k \geq 1$. We introduce a class of formulas φ over variables in $G \cup X \cup \{pc, t_1, \dots t_k\}$, where the *thread variables* $t_1, \dots, t_k$ range over the set $Tid = \{1, \dots, n\}$ of thread indices (which are at the same time thread ids). A local variable $x \in X$, a local variable pc, and a thread variable t_i may occur in φ only in an *array read* of the form $x[t_i]$ or $pc[t_i]$. In particular, the thread variables t_i may occur only as the index of an array read.

An *array invariant of width k* is an array invariant Ψ of the form

$$\Psi \ \equiv\ \forall t_1, \dots, t_k.\ \mathit{Distinct}(t_1, \dots, t_k) \rightarrow \varphi$$

where φ is a formula in the class given above. Thus, Ψ has k universally quantified thread variables; its free variables lie in $G \cup X \cup \{pc\}$.

An *Ashcroft invariant of width k* for $P(n)$ is defined as an array invariant for $P[n]$ of width k. It is called a *uniform* Ashcroft invariant for P of width k if it is an Ashcroft invariant for $P(n)$ for every $n \geq k$.

Proposition 1. *If $n \geq k$, then the existence of an Ashcroft invariant of width k for $P(n)$ (the concurrent program with n threads) implies that $P(n)$ is correct. The existence of a uniform Ashcroft invariant of width k for P (the parameterized program) implies that P is correct, i.e., that $P(n)$ is correct for every $n \geq 1$.*

A uniform Ashcroft invariant of width k for P is generally not a *safe* inductive invariant for $P(n)$ for $n < k$ (in fact, it is equivalent to *true* for $n < k$). The second part of the proposition above still holds because the safety of a program $P(k)$ (trivially) implies the safety of all programs $P(n)$ for all $n < k$.

Cartesian Abstraction. Intuitively, Cartesian abstraction *abstracts away* the dependence between the components of a tuple by looking at only one component at a time. The extension to k-Cartesian abstraction is to look at k components at a time.

The connection between thread modularity and Cartesian abstraction is well-known; see, e.g., [16, 17, 42]. In the setting of concurrent programs, Cartesian abstraction looks at the local state of one thread at a time (together with the global state).

In the pure version of Cartesian abstraction as used, e.g., in [9], the abstraction function α maps a set X of n-tuples of values to an n-tuple $(Y_1, \dots, Y_n)$ of sets of values, and the concretization function γ maps an n-tuple of sets $(Y_1, \dots, Y_n)$ to their Cartesian product. Then $\gamma(\alpha(X))$ is the smallest set in the form of a Cartesian product that contains X as a subset. If the set X of n-tuples is symmetric (closed under permutation), we take α as a function that maps X simply to a set Y of values (as opposed to the n-tuple $(Y, \dots, Y)$ with the same set Y in every component). In our extension, α maps X to a set Y of k-tuples of values.

Definition 2 (k-Cartesian Abstraction)**.** *The k-Cartesian abstraction for $P(n)$ is defined by the following abstraction function α_k which maps a set X of states of $P(n)$ to a set of states of $P(k)$.*

$$\begin{aligned}\alpha_k(X) = \{((\ell_{t_1}, \dots, \ell_{t_k}), \nu_k) \mid\ & ((\ell_1, \dots, \ell_n), \nu) \in X, \\ & \mathit{Distinct}(t_1, \dots, t_k), \\ & \nu_k(g) = \nu(g), \\ & \nu_k(x_1) = \nu(x_{t_1}) \\ & \vdots \\ & \nu_k(x_k) = \nu(x_{t_k}) \qquad \}.\end{aligned}$$

By the theory of *abstract interpretation* [15], the concretization function γ_k (which maps a set Y of states of $P(k)$ to a set of states of $P(n)$) is determined by $\gamma_k(Y) = \bigcup\{X \mid \alpha_k(X) \subseteq Y\}$. Equivalently,

$$\gamma_k(Y) = \{((\ell_1, \dots, \ell_n), \nu) \mid \alpha_k(\{((\ell_1, \dots, \ell_n), \nu)\}) \subseteq Y\}.$$

We now define what it means that *the k-Cartesian abstraction of the concurrent program $P(n)$ is safe*, for $n \geq 1$. We cannot define the Cartesian abstraction of a program by a source-to-source transformation of the program or by an abstract transition system (there is no notion of *abstract state*). Instead, we will use the post operator of $P(n)$ which is defined by

$$\mathit{post}(X) = \mathsf{init} \cup \{\sigma' \mid \exists \sigma \in X.(\sigma, \sigma') \text{ is a transition of } P(n)\}$$

for a set of states X of $P(n)$. The abstract post operator in the k-Cartesian abstraction $P(n)$ is defined by $\mathit{post}^{\#} = \alpha_k \circ \mathit{post} \circ \gamma_k$. We say that the k-Cartesian abstraction of $P(n)$ is safe if $\gamma_k(\mathit{post}^{\#^m}(\emptyset)) \cap \mathsf{err}_n = \emptyset$ for all m ("no error state is reachable in the abstract").

Lemma 2. *If there exists an Ashcroft invariant of width k for $P(n)$, then the k-Cartesian abstraction of $P(n)$ is safe (for $n \geq k$).*

Proof. Let $\Psi = \forall t_1, \dots, t_k \in Tid.\ \mathit{Distinct}(t_1, \dots, t_k) \rightarrow \varphi$ be an Ashcroft invariant for $P(n)$. We define as X_Ψ the states of $P(n)$ that satisfy Ψ, to be accurate, the set of states that are bisimilar to a state of $P[n]$ that satisfies Ψ. Using the bisimulation between $P(n)$ and $P[n]$ and the fact that Ψ is an array invariant of $P[n]$, it follows that X_Ψ contains the initial states of $P(n)$, is closed under the transition relation and disjoint from the error states, i.e., $\mathit{post}(X_\Psi) \subseteq X_\Psi$ and $X_\Psi \cap \mathsf{err}_n = \emptyset$.

Similarly, we define Y_φ as the set of states of $P(k)$ that satisfy the instantiation of φ to threads $1, \ldots, k$, to be accurate, the set of states that are bisimilar to a state of $P[k]$ that satisfies $\varphi[1/t_1] \ldots [k/t_k]$. By definition of Ψ, we have $X_\Psi = \gamma_k(Y_\varphi)$ and $\alpha_k(X_\Psi) \subseteq Y_\varphi$.

Now, $\gamma_k(Y_\varphi) = X_\Psi$ and $post(X_\Psi) \subseteq X_\Psi$ imply

$$post^\#(Y_\varphi) = \alpha_k(post(X_\Psi)) \subseteq \alpha_k(X_\Psi) \subseteq Y_\varphi.$$

Hence, $post^{\#^{m+1}}(\emptyset)) \subseteq post^{\#^{m+1}}(Y_\varphi) \subseteq Y_\varphi$. Finally, we have $\gamma_k(post^{\#^{m+1}}(\emptyset)) \subseteq \gamma_k(Y_\varphi) = X_\Psi$. Since $X_\psi \cap \mathsf{err}_n = \emptyset$, this shows that the k-Cartesian abstraction is safe. □

From now, for technical reasons, we assume that k is not smaller than the dimension m of the generator set of error states (defined in Section 2).

Theorem 4 (Relative Completeness)**.** *Given a parameterized program P, the following are equivalent.*

1. *P has a k-thread-modular proof.*
2. *There is a uniform Ashcroft invariant of width k for P.*
3. *There is an Ashcroft invariant of width k for $P(k+1)$.*
4. *The k-Cartesian abstraction of $P(k+1)$ is safe.*

Proof. The implication $(1) \Rightarrow (2)$ follows from Lemma 3 in Section 5.1. For $(2) \Rightarrow (3)$, note that a uniform Ashcroft invariant for P is an Ashcroft invariant for $P(n)$ for every $n \geq k$. The implication $(3) \Rightarrow (4)$ follows from Lemma 2 with $n = k+1$. For $(4) \Rightarrow (1)$, see Lemma 4 in Section 5.2. □

The equivalence of $(3) \Leftrightarrow (2)$ means that if there is an Ashcroft invariant of width k for $P(k+1)$, there is also a uniform Ashcroft invariant of width k for P. We note a similar consequence of Theorem 4: if the k-Cartesian abstraction of $P(k+1)$ is safe, the k-Cartesian abstraction of $P(n)$ is also safe for any $n \geq k$. This follows from the implication $(4) \Rightarrow (2)$ and Lemma 2.

5.1 From Level k to k Quantifiers

We now show that thread-modular proofs at level k are a sound proof method by giving a uniform Ashcroft invariant with k universal quantifiers.

Lemma 3. *If the annotation Φ for a program $P(k)$ is a thread-modular proof at level k, then*

$$\Psi := \forall t_1, \ldots, t_k \in \mathit{Tid}.\ \mathit{Distinct}(t_1, \ldots, t_k) \rightarrow \bigwedge_{(\ell_1,\ldots,\ell_k)\in\mathsf{Loc}^k} ((\bigwedge_{i=1}^{k} pc[t_i] = \ell_i) \rightarrow \Phi(\ell_1, \ldots, \ell_k)[x[t_1]/x_1] \cdots [x[t_k]/x_k])$$

is a uniform Ashcroft invariant for P.

Proof. Let $n \geq k$ and $\mathit{Tid} = \{1, \ldots, n\}$. The initialization property of Ashcroft invariants follows directly from **(initial)** in Definition 1.

To show the continuation property of an Ashcroft invariant, we pick an arbitrary $(\ell_{src}, s, \ell_{tgt}) \in \delta$ and $t \in \mathit{Tid}$ and show the validity of

$$\{\Psi\}\, [pc[t] = \ell_{src}]; s[t]; pc[t] := \ell_{tgt}\, \{\Psi\}$$

In the postcondition, we fix distinct $t_1, \ldots, t_k \in \mathit{Tid}$ and $(\ell_1, \ldots, \ell_k) \in \mathsf{Loc}^k$ and show that

$$(\bigwedge_{i=1}^{k} pc[t_i] = \ell_i) \rightarrow \Phi(\ell_1, \ldots, \ell_k)[x[t_1]/x_1] \cdots [x[t_k]/x_k] \quad (1)$$

holds after executing the program fragment on a state satisfying Ψ.

***Case 1:** $t = t_i$ **for some** $i = 1, \ldots, k$.* If the pre-state does not satisfy $pc[t_j] = \ell_j$ for some $j \neq i$ or if $\ell_i \neq \ell_{tgt}$, the formula (1) holds in the post-state because the left side of the implication is false. If $pc[t_i] = \ell_{src}$ does not hold in the pre-state, the assumption in the program fragment fails and the Hoare triple is valid. Otherwise, Ψ and therefore also

$$\Phi(\ell_1, \ldots, \ell_{i-1}, \ell_{src}, \ell_{i+1}, \ldots, \ell_k)[x[t_1]/x_1] \cdots [x[t_k]/x_k]$$

hold in the pre-state. One needs to show that after executing $s[t]$ the post-state satisfies

$$\Phi(\ell_1, \ldots, \ell_{i-1}, \ell_{tgt}, \ell_{i+1}, \ldots, \ell_k)[x[t_1]/x_1] \cdots [x[t_k]/x_k]\,.$$

This follows from the property **(inductive)** in Definition 1.

***Case 2:** t **is distinct from** $t_1, \ldots, t_k$.* If the pre-state does not satisfy $pc[t_j] = \ell_j$ for some $j = 1, \ldots, k$, the formula (1) holds in the post-state since the left side of the implication is false. If $pc[t] = \ell_{src}$ does not hold in the pre-state, the assumption in the program fragment fails and the Hoare triple is valid. Otherwise, Ψ and therefore also

$$\Phi(\ell_1, \ldots, \ell_k)[x[t_1]/x_1] \cdots [x[t_k]/x_k]$$

hold. By instantiating t_i in Ψ with t for $i = 1, \ldots, k$, we can also derive that

$$\Phi(\ell_1, \ldots, \ell_{i-1}, \ell_{src}, \ldots \ell_k)[x[t_1]/x_1] \cdots [x[t]/x_i] \cdots [x[t_k]/x_k]$$

holds in the pre-state. One needs to show that after executing $s[t]$ the post-state satisfies

$$\Phi(\ell_1, \ldots, \ell_k)[x[t_1]/x_1] \cdots [x[t_k]/x_k]\,.$$

This follows from the property **(non-interference)** in Definition 1.

Now consider for a location $(\ell_1, \ldots, \ell_n)$ the formula

$$\mathsf{err}(\ell_1, \ldots, \ell_n) = \bigwedge_{1 \leq i_1 < \cdots < i_m \leq n} \mathsf{err}(\ell_{i_1}, \ldots, \ell_{i_m})\,.$$

We assume that $pc[j] = \ell_j$ for $1 \leq j \leq n$, that $\mathsf{err}(\ell_{i_1}, \ldots, \ell_{i_m})$ for some $1 \leq i_1 < \cdots < i_m \leq n$ holds and that Ψ holds and show that this leads to a contradiction. For the case $k \geq m$, set $t_j = i_j$ for $j \leq m$ and pick distinct t_j arbitrarily for $m < j \leq k$. Since Ψ holds, also $\varphi(\ell_{t_1}, \ldots, \ell_{t_k})$ holds. Since $t_j = i_j$ for $j \leq m$, also $\mathsf{err}(\ell_{t_1}, \ldots, \ell_{t_m})$ holds. Using the property **(safe)** in Definition 1 we can derive the contradiction. For the case $k < m$, instantiate $t_1 < \cdots < t_k$ in Ψ for every subset $\{t_1, \ldots, t_k\} \subseteq \{i_1, \ldots, i_m\}$. Again, the property **(safe)** in Definition 1 can be used to derive a contradiction. □

5.2 From k-Cartesian to Level k

The following lemma relies on our assumption k is not smaller than the dimension m of the generator set of error states.

Lemma 4. *If the k-Cartesian abstraction of $P(k+1)$ is safe then P has a k-thread-modular proof.*

Proof. We show the contraposition using Lemma 1: if there is a derivation of *false* in the Horn clause system, then the k-Cartesian abstraction is not safe.

In particular, we show that if there is a derivation of depth m in the Horn clause system of $\mathit{Inv}(g, \ell_1, x_1, \ldots, \ell_k, x_k)$, then there is a state $((\ell_1, \ldots, \ell_k, \ell_{k+1}), \nu)$ of $P(k+1)$ with $\nu(x_i) = x_i$ for $i = 1, \ldots, k$ that is reachable in at most m steps in the k-Cartesian abstraction, i. e.,

$$((\ell_1, \ldots, \ell_k, \ell_{k+1}), \nu) \in \gamma_k(post^{\#^m}(\emptyset)).$$

We prove this by induction over m.

For $m = 1$, $\mathit{Inv}(g, \ell_1, x_1, \ldots, \ell_k, x_k)$ is derived from the clause (initial), i.e., $\ell_1 = \cdots = \ell_k = \ell_{\mathsf{init}}$ and $x_i \in \mathsf{init}_{\mathsf{local}}$.

For ν with $\nu(x_i) = x_i$ and $\nu(x_{k+1}) = \nu(x_1)$, we have $((\ell_1, \ldots, \ell_k, \ell_{init}), \nu) \in \mathsf{init} \subseteq \gamma_k(post^\#(\emptyset))$.

$m \mapsto m+1$: If the last derivation step was an inductive step

$$Inv(g, \ell_1, \ldots, \ell_i, x_i, \ldots, x_k) \to Inv(g', \ell_1, \ldots, \ell_i', x_i', \ldots, x_k)$$

then by induction hypothesis there is a state

$$((\ell_1, \ldots, \ell_k, \ell_{k+1}), \nu) \in \gamma_k(post^{\#^m}(\emptyset))$$

where $\nu(g) = g, \nu(x_j) = x_j$ for $j \in \{1, \ldots, k\}$. From this state, $P(k+1)$ can take the same edge in the i-th thread and reach $((\ell_1, \ldots, \ell_i', \ldots, \ell_k, \ell_{k+1}), \nu')$ where $\nu'(g) = g'$, $\nu'(x_i) = x_i'$ and $\nu'(x_j) = \nu(x_j)$ for $j \neq i$. Thus, this state is in

$$post(\gamma_k(post^{\#^m}(\emptyset))) \subseteq \gamma_k(post^{\#^{m+1}}(\emptyset)).$$

If the last derivation step uses the non-interference clause from Figure 6, then for $Inv(g, \ell_1, x_1, \ldots, \ell_k, x_k)$ we have by induction hypothesis

$$((\ell_1, \ldots, \ell_k, \ell_{k+1}), \nu) \in \gamma_k(post^{\#^m}(\emptyset))$$

where $\nu(g) = g, \nu(x_i) = x_i$. Hence, projecting on the first k variables, we have

$$((\ell_1, \ldots, \ell_k), \nu') \in \alpha_k(\gamma_k(post^{\#^m}(\emptyset)))$$

where $\nu'(x_i) = \nu(x_i) = x_i$ for $i = 1, \ldots, k$, and similar for all permutations of the k threads. From the other literals on the left side of the Horn clause, we also get

$$((\ell_1, \ldots, \ell_{i-1}, \ell^\dagger, \ell_{i+1} \ldots, \ell_k), \nu_i) \in \alpha_k(\gamma_k(post^{\#^m}(\emptyset)))$$

where $\nu_i(x_i) = x^\dagger$ and $\nu_i(x_j) = x_j$ for $j \neq i$. By definition of the abstraction function and since $\gamma_k \circ \alpha_k \circ \gamma_k = \gamma_k$, we obtain

$$((\ell_1, \ldots, \ell_k, \ell^\dagger), \nu) \in \gamma_k(post^{\#^m}(\emptyset))$$

with $\nu(x_i) = x_i, i = 1, \ldots, k$ and $\nu(x_{k+1}) = x^\dagger$ since all its projections to k threads are in $\alpha_k(\gamma_k(post^{\#^m}(\emptyset)))$. This state has the successor $((\ell_1, \ldots, \ell_k, \ell^{\dagger\prime}), \nu')$ where $\nu'(g) = g'$ and $\nu'(x_i) = x_i$ for $i = 1, \ldots, k$. Thus, this state is in

$$post(\gamma_k(post^{\#^m}(\emptyset))) \subseteq \gamma_k(post^{\#^{m+1}}(\emptyset))$$

and the induction thesis holds for $m+1$.

Finally, if *false* is derived, it can only be derived by the safe clause from Figure 6. The induction hypothesis

$$((\ell_1, \ldots, \ell_k, \ell_{k+1}), \nu) \in \gamma_k(post^{\#^m}(\emptyset))$$

holds for ν with $\nu(g) = g$, $\nu(x_i) = x_i, i = 1, \ldots, k$. Also, $\mathsf{err}(g, \ell_1, x_1, \ldots, \ell_m, x_m)$ holds. By definition of err_n, this implies that $\nu \in \mathsf{err}_{k+1}(\ell_1, \ldots, \ell_{k+1})$. Hence, the k-Cartesian abstraction of $P(k+1)$ is not safe. □

5.3 Finite-state Threads

We conclude this section with an application of Theorem 4, which shows that k-thread-modular proofs are complete for parameterized programs with *finite-state threads*; i.e., when the global and local variables in the underlying thread template range over finitely many values.

Theorem 5. *A parameterized program P with finite-state threads is correct if and only if there is a level k such that P has a thread-modular proof at level k.*

Proof. The "if" direction follows by soundness. We prove the "only if" direction. We formulate the problem of checking if P is correct as the *non-coverability* problem for a *well-structured transition system* [2]. We do it in two steps as follows.

First, we maintain a counter for each control location and each valuation of the local variables. The counter for a location and a valuation tracks the number of threads at that control location and with that valuation. With this encoding, a state consists of a finite part giving the valuation to the global variables and a finite vector of non-negative counters. The set of states is *well-quasi ordered* (see [2] for the basic definitions), where one state is less than equal to another iff their valuations to the global variables are identical and the vectors of counters of the first is less than equal to the other co-ordinatewise. We lift the transitions of the program P to transitions between states in the natural way. The transition relation is *monotonic* with respect to the well-quasi ordering; this is because adding more threads to a program does not prohibit any transition that was allowed before. This encoding shows that the program P corresponds to a well-structured transition system.

Second, since the set of error states is representable as an assertion with only existential quantifiers, it determines an upward closed set of states in the well-quasi ordering of states. Thus, checking if P is correct is equivalent to checking that the upward closed set of error states is not reachable from the initial states. This is the non-coverability problem for the well-structured transition system defined by P.

Let B (for "Bad") be the set of states which can reach the upward closed set of error states. From the theory of well-structured systems [2], we know that B is also upward closed and representable as the upward closure of a finite set of tuples, called the *minimal elements*, where each minimal element consists of a valuation to the global variables and a tuple of natural numbers k_{ij}, giving a count of the minimal number of threads at location ℓ_i and with local state j.

Given a representation of B by the finite set E of its minimal elements, let k be the maximum of all k_e, where k_e is the sum of all numbers k_{ij} appearing in a minimal element $e \in E$. Note that k is finite because E is finite. Let $\neg B$ be the complement of B. We write $B(n)$ (resp. $\neg B(n)$) for the set of states of $P(n)$, for $n \geq 1$, that are encoded by states in B (resp. $\neg B$) and $B(\geq k)$ (resp. $\neg B(\geq k)$) for the set of states $\cup_{n \geq k} B(n)$ (resp. $\cup_{n \geq k} \neg B(n)$).

By definition, $\neg B(n)$ does not intersect with the set of error states, and it is closed under transitions of $P(n)$. By assumption, P is correct, thus $P(n)$ is correct, for every $n \geq 1$. Thus, $\neg B(n)$ contains the set of initial states of $P(n)$, for every $n \geq 1$. Because of this, $\neg B(n)$ defines a safe inductive invariant for $P(n)$.

We show that $B(\geq k)$, the restriction of B to the set of programs with at least k threads, has a representation as a *finite* disjunction of assertions with k existential quantifiers. Essentially, the disjunction is over the set of minimal elements in the representation of B, and each disjunct encodes the fact that the global variables evaluate to the value given by the minimal element and that there are at least k_{ij} distinct threads in location ℓ_i with valuation j to the local variables. For example, the minimal element "the value of the global variable g is c_g and there are at least k_1 threads in location ℓ_1 with value c_{x1} for the local variable and at least k_2 threads in ℓ_2 with value c_{x2}" is expressible as

$$\begin{aligned} &\exists t_1 \exists t_2 \ldots \exists t_{k_1} \exists t_{k_1+1} \ldots \exists t_{k_1+k_2} \in Tid. \\ &\quad Distinct(t_1, \ldots, t_{k_1}, t_{k_1+1}, \ldots, t_{k_1+k_2}) \wedge g = c_g \wedge \\ &\quad \bigwedge_{j=1}^{k_1} (pc[t_j] = \ell_1 \wedge x[t_j] = c_{x1}) \wedge \\ &\quad \bigwedge_{j=k_1+1}^{k_1+k_2} (pc[t_j] = \ell_2 \wedge x[t_j] = c_{x2}) \end{aligned}$$

To accommodate the set of all minimal elements, we first existentially quantify k thread identifiers, assert they are all distinct, and then take the disjunction of the constraints for each minimal element. In summary, $B(\geq k)$, the restriction of B to the set of encodings of states of $P(n)$ with $n \geq k$, can be expressed by a formula with k existential quantifiers. Now, for each n with nk, the set $B(n)$, the restriction of B to the set of encodings of states of $P(n)$, can be expressed by that same formula. Then $\neg B(n)$ has a representation as an Ashcroft invariant of width k, for $n \geq k$. Since the Ashcroft invariant does not depend on n, we have a *uniform* Ashcroft invariant for P.

By Theorem 4 ("2 $\Rightarrow$ 1"), since there is a uniform Ashcroft invariant for P of width k, there will also exist a k-thread-modular proof for P. □

A similar completeness result has been noted in the context of algorithms for parameterized verification [4, 32]. The arguments there directly use the well-quasi ordering to argue termination of a specific algorithm and do not consider provability in a proof system. Since Petri net coverability is EXPSPACE-hard [41], and Petri nets can be simulated by our model, this also means that some programs can be proved correct only using thread-modular proofs at level k where k is doubly exponential in the size of the template.

6. Mach TLB Shootdown

We now present a 2-thread-modular proof for the TLB shootdown algorithm from [13]. In this algorithm, one thread can send interrupts and messages to the other threads. To support this feature, we extend our programming model by adding a new kind of statement, called *synchronous statement*, which allows one thread to read and write local variables of a second thread (Section 6.2). We modify our proof rules to accommodate synchronous statements. Finally, we present a 2-thread-modular proof for the algorithm.

6.1 TLB Shootdown Algorithm

We first introduce the algorithm informally. The TLB shootdown algorithm [13] is part of the Mach operating system. Modern processors use page tables to translate between virtual and physical memory. For performance reasons, they cache the page table in a Translation Look-aside Buffer (TLB). The problem is to keep this cache in sync if the page table needs to be updated. While typical first and second-level caches have a cache coherence protocol implemented in hardware, there is no hardware support to keep the TLB consistent.

In [13], the authors present a software solution to achieve consistency. The idea is that before one processor updates the page table, it interrupts all other processors that use the page table. Then it waits until the interrupt was received by the other processors and updates the page table. The other processors wait until the first processor is done and then they flush the TLB.

A TLB update (see Figure 9) proceeds in four phases. In the first phase the INITIATOR is called on the page map (pmap, in short) that should be updated. The initiator runs lines 1 to 8 to set the *actionneeded* flag and signals an interrupt to all processors that use the pmap (indicated by $userpmap[cpu] = pmap$). Then it waits for these CPUs to respond. In the second phase, the other CPUs receive the interrupt and invoke the RESPONDER. They set the active flag to false (indicating they are done with using the page map) and wait at line 3. The INITIATOR waits until all responder processes have set their active flag to false or are not using the pmap (lines 9-10). Then, it continues in phase 3. In phase 3 the INITIATOR updates the pmap entry and its tlb (line 11 to 14). When the INITIATOR finishes with phase 3, the responders flush their tlb (line 5 of RESPONDER) and reset the *actionneeded* flag (line 6). There are some other special cases that this code handles correctly, e.g., when two processors run as INITIATOR and when each processor updates the pmap of the other processor.

Our code abstracts from the original Mach pseudo code. First we assume that each processor has only one pmap, while in Mach

```
INITIATOR($pmap$)
 1  $active := false$
 2  lock $plock[pmap]$
 3  for $cpu \neq pid$
 4       do if $userpmap[cpu] = pmap$
 5            then lock $actionlock[cpu]$
 6                 $actionneeded[cpu] := true$
 7                 unlock $actionlock[cpu]$
 8                 $interrupt[cpu] := true$
 9  for $cpu \neq pid$
10       do wait until $\neg active[cpu]$
                 $\vee\ usermap[cpu] \neq pmap$
11  $entry[pmap] := *$
12  if $userpmap = pmap$
13     then $tlb := entry[pmap]$
14  unlock $plock[pmap]$
15  $active := true$

RESPONDER()
1  while $actionneeded$
2       do $active := false$
3          wait until $plock[userpmap] = unlocked$
4          lock $actionlock$
5          $tlb := entry[userpmap]$
6          $actionneeded := false$
7          unlock $actionlock$
8          $active := true$

MAIN()
1  while $true$
2       do assert $tlb = entry[userpmap]$
3          if $*$
4            then INITIATOR($*$)
5          if interrupt
6            then $interrupt := 0$; RESPONDER()

BOOT()
1  $pid := ++n$
2  atomic $userpmap := *$; assume $1 \leq userpmap \leq n$;
          assume $plock[userpmap] = unlocked$;
          $tlb := entry[userpmap]$
3  MAIN()
```

Figure 9. The TLB shootdown algorithm. In the correct version lines 3 and 4 of RESPONDER must be executed atomically.

it has two (kernel and user pmap). Secondly, our page table is modeled by a single entry and the TLB is modeled by a cache of this single entry. A flush is modeled as reading the entry (line 5 of RESPONDER), while in reality the process only marks the entry as invalid and rereads it only when it is accessed. The changes enable us to model the condition when an error occurs by a simple assertion but they do not affect the complicated synchronization scheme between the processors.

The main correctness property is given in line 2 of the main procedure. Whenever the pmap entry is changed in line 11, the protocol ensures that the tlb will be flushed, either by line 13 of the initiator or line 5 of the responder before line 2 can be executed.

In our settings the processors are threads and all execute the same thread template (the procedure MAIN). Each thread has a local variable pid between 1 and n that stores its own processor id. We achieve this by adding a small initialization procedure BOOT.

This procedure uses a global variable n that denotes the number of initialized processes. On initialization, as in Example 2, this global variable is atomically incremented (we abbreviate this by $++n$ in the code) and assigned to the local variable pid. This ensures that after this first instruction runs all processes have a distinct process id between 1 and n. We initialize pid to 0 in the init assertion to ensure that an uninitialized processes cannot have the same pid as an initialized process.

The loops "**for** $cpu \neq pid$" are implemented as loops from 1 to the global variable n that skip over the body if $cpu = pid$ holds. It can happen that another process is initialized after the loop is finished. In that case the initiator is not able to notify this process. However, the boot routine of the newly initialized process will ensure that the pmap is unlocked and initializes the TLB entry atomically.

We model the arrays by local variables that reside in other threads. Reading or writing to these variables requires a *synchronous statement*, that we describe in Section 6.2. For example, the instruction **lock** $actionlock[cpu]$ is translated to the synchronous statement

$$[pid_2 = cpu_1 \wedge actionlock_2 = false]; actionlock_2 := true\,.$$

This statement mentions variables cpu_1 from the thread executing the statement and variables pid_2 and $actionlock$ from another thread. The assumption $[pid_2 = cpu_1]$ ensures that the instruction is executed against the thread cpu_1. The remaining part implements the atomic lock mechanism.

6.2 Synchronous Statements

A synchronous statement s affects the local variables of two threads at the same time. That is a synchronous statement uses two copies x_1 and x_2 of the local variables. For simplicity we do not allow to write to global variables in synchronous statements. These statements are executed by the first thread (with the local variables x_1) and may also change the local variables x_2 of the second thread but not its program counter.

For a synchronous statement s, we define $(s : i, j)$ as the statement obtained from s by replacing variables x_1 by x_i and variables x_2 by x_j. Then we extend the semantics of the parameterized process $P(n)$ by adding an edge

$$((\ell_1, \dots, \ell_{i-1}, \ell_{src}, \ell_{i+1}, \dots, \ell_n), \\ (s : i, j), (\ell_1, \dots, \ell_{i-1}, \ell_{dst}, \ell_{i+1}, \dots, \ell_n))$$

for every $i \neq j$, $1 \leq i, j \leq n$ whenever $(\ell_{src}, s, \ell_{dst})$ is an edge in the template. That is for every thread $j \neq i$, an edge with the statement $(s : i, j)$ is added.

To check thread modularity at level k in the presence of synchronous statements we have to adapt the rules **(inductive)** and **(non-interference)**. The new rule **(inductive)** is straightforward. For **(non-interference)**, we only need to prove that the statement preserves the invariant for the case that it is between thread $k + 1$ and some thread i, $1 \leq i \leq k$.

Definition 3. For programs with synchronous statements the rules inductive and non-interference from Definition 1 are changed as follows.

(inductive) The assertions φ and ψ at the source resp. target location of an edge $(\ell_{src}, s, \ell_{dst})$ in $P(k)$ and the statement labeling the edge form a valid Hoare triple. That is, for the assertion $\varphi = \Phi(\ell_1, \dots, \ell_{i-1}, \ell_{src}, \ell_{i+1}, \dots, \ell_k)$ and the assertion $\psi = \Phi(\ell_1, \dots, \ell_{i-1}, \ell_{dst}, \ell_{i+1}, \dots, \ell_k)$

$$\{\varphi\}\ (s : i, j)\ \{\psi\}$$

is valid for $P(k)$ and $1 \leq i, j \leq k$, $i \neq j$.

(non-interference) Every assertion φ labeling a location $(\ell_1, \dots, \ell_k)$ in $P(k)$ is interference-free under the execution of a synchronous statement by thread $(k + 1)$. That is, for the assertion $\varphi = \Phi(\ell_1, \dots, \ell_k)$ and $\psi_1 = \Phi(\ell_{src}, \ell_2, \dots, \ell_k)$, $\dots$, $\psi_k = \Phi(\ell_1, \dots, \ell_{k-1}, \ell_{src})$

$$\{\varphi \wedge \psi_1[x_{k+1}/x_1] \wedge \dots \wedge \psi_k[x_{k+1}/x_k]\}\ (s : k + 1, j)\ \{\psi\}$$

is valid for $P(k + 1)$ for $1 \leq j \leq k$ and each labeled edge $(\ell_{src}, s, \ell_{dst})$.
Thread $k + 1$ can also passively interfere if thread $i \leq k$ executes a synchronous statement affecting thread $k + 1$. Thus, we require that for the assertion $\varphi = \Phi(\ell_1, \dots, \ell_{src}, \dots, \ell_k)$, $\psi = \Phi(\ell_1, \dots, \ell_{dst}, \dots, \ell_k)$, and $\psi_1 = \Phi(\ell_{k+1}, \ell_2, \dots, \ell_k)$, $\dots$, $\psi_k = \Phi(\ell_1, \dots, \ell_{k-1}, \ell_{k+1})$,

$$\{\varphi \wedge \psi_1[x_{k+1}/x_1] \wedge \dots \wedge \psi_k[x_{k+1}/x_k]\}\ (s : i, k + 1)\ \{\psi\}$$

is valid for $P(k + 1)$ for $1 \leq i \leq k$ and each labeled edge $(\ell_{src}, s, \ell_{dst})$.

6.3 Correctness

The implementation as given in Figure 9 has a computation that reaches the error state. The problem is that the step from line 3 to 4 in RESPONDER is not atomic. If another process acting as initiator runs through line 2 to 5 in INITIATOR and takes the action lock before the responder moves from line 3 to 4 in RESPONDER, the system reaches an inconsistent state that may eventually lead to the error. In personal communication with one of the original authors of the protocol [12], we learned that the authors considered the algorithm under an (unstated) relative-speed assumption on the hardware on which the algorithm was supposed to run. Because the protocol runs with interrupts disabled on real separated processors, in practice, an INITIATOR process cannot perform all these steps before the RESPONDER process is able to take the lock. However, this relative speed assumption need not hold on virtual machines where CPUs are simulated and this bug may occur.

To fix this issue, we changed the implementation to execute lines 3 and 4 of RESPONDER atomically. We also augmented the waiting condition on line 3 to check that the $actionlock$ is not taken when line 3 is executed. After this change, the system is provably correct. Our thread-modular proof on level 2 uses the invariant in Figure 10. The proof that the invariant is inductive and interference-free was machine-checked using Z3.

7. Related Work

Thread-modular proof systems are the cornerstone of concurrency verification [1, 8, 28, 37, 47]. In this paper, we have investigated a hierarchy of proof systems generalizing thread modularity and characterized the power of these proofs in concurrent program verification. We now put our work into perspective, pointing out how similar ideas have been studied, often implicitly and under different names. Similar generalizations to thread modularity have been studied in various guises [4, 5, 14, 30, 33, 35, 45, 51–53]. For example, 2-thread-modular proofs were studied in [14, 45], under the name pairwise invariants or split invariants, as context inference in [33], and as thread correlation analysis in [52]. In the context of parameterized verification of finite-state processes, *view abstraction* [4, 5] presents similar notions as an abstract interpretation; completeness of view abstraction for finite-state threads follows through a reasoning based on well-structured transition systems.

In the setting of Horn clauses, Grebenshchikov et al. [30], Hojjat et al. [35], Gurfinkel and Shoham [32], and Monniaux [44] study Horn constraints of the kind that we use to encode k-thread-modular proofs; in particular, [35] already note the strictness of the corresponding hierarchy of proofs. The main emphasis in these

$$
\begin{aligned}
\Phi = {} & (pc_1 = \textsc{Boot}_1 \wedge pid_1 = 0) \vee \\
& (1 \leq pid_1 \leq n \wedge pid_1 \neq pid_2 \wedge (1 \leq userpmap_1 \leq n \vee pc_1 = \textsc{Boot}_2) \\
& \wedge (\textsc{Init}_2 < pc_1 \leq \textsc{Init}_{14} \wedge pid_2 = pmap_1 \rightarrow plock_2) \\
& \wedge (\textsc{Init}_2 < pc_1 \leq \textsc{Init}_{14} \wedge \textsc{Init}_2 < pc_2 \leq \textsc{Init}_{14} \rightarrow pmap_1 \neq pmap_2) \\
& \wedge (\textsc{Init}_5 < pc_1 \leq \textsc{Init}_7 \wedge pid_2 = cpu_1 \rightarrow actionlock_2) \\
& \wedge (\textsc{Resp}_4 < pc_1 \leq \textsc{Resp}_7 \rightarrow actionlock_1) \\
& \wedge (\textsc{Init}_5 < pc_1 \leq \textsc{Init}_7 \wedge \textsc{Init}_5 < pc_2 \leq \textsc{Init}_7 \rightarrow cpu_1 \neq cpu_2) \\
& \wedge (\textsc{Init}_5 < pc_1 \leq \textsc{Init}_7 \wedge \textsc{Resp}_4 < pc_2 \leq \textsc{Resp}_7 \rightarrow pid_2 \neq cpu_1) \\
& \wedge (\textsc{Init}_4 \leq pc_1 \leq \textsc{Init}_8 \rightarrow cpu_1 \neq pid_1) \\
& \wedge (\textsc{Init}_5 \leq pc_1 \leq \textsc{Init}_8 \wedge pid_2 = cpu_1 \rightarrow userpmap_2 = pmap_1) \\
& \wedge (\neg active_1 \rightarrow \textsc{Init}_1 < pc_1 \leq \textsc{Init}_{15} \vee \textsc{Resp}_2 < pc_1 \leq \textsc{Resp}_8) \\
& \wedge (\textsc{Init}_7 \leq pc_1 \leq \textsc{Init}_8 \wedge userpmap_2 = pmap_1 \wedge pid_2 = cpu_1 \rightarrow actneed) \\
& \wedge (\textsc{Init}_3 \leq pc_1 \leq \textsc{Init}_8 \wedge userpmap_2 = pmap_1 \wedge pid_2 < cpu_1 \rightarrow actneed) \\
& \wedge (\textsc{Init}_9 \leq pc_1 \leq \textsc{Init}_{14} \wedge userpmap_2 = pmap_1 \rightarrow actneed) \\
& \wedge (\textsc{Init}_3 \leq pc_1 \leq \textsc{Init}_8 \wedge userpmap_2 = pmap_1 \wedge pid_2 < cpu_1 \rightarrow interrupted) \\
& \wedge (\textsc{Init}_9 \leq pc_1 \leq \textsc{Init}_{14} \wedge userpmap_2 = pmap_1 \rightarrow interrupted) \\
& \wedge (\textsc{Init}_3 \leq pc_1 \leq \textsc{Init}_8 \wedge userpmap_2 = pmap_1 \wedge pid_2 < cpu_1 \rightarrow interrupted) \\
& \wedge (\textsc{Init}_9 \leq pc_1 \leq \textsc{Init}_{10} \wedge userpmap_2 = pmap_1 \wedge pid_2 < cpu_1 \rightarrow inactive) \\
& \wedge (\textsc{Init}_{11} \leq pc_1 \leq \textsc{Init}_{14} \wedge userpmap_2 = pmap_1 \rightarrow inactive) \\
& \wedge (userpmap_2 = pid_1 \wedge tlb_2 \neq entry_1 \rightarrow dirty \vee \textsc{Boot}_1 \leq pc_2 \leq \textsc{Boot}_2) \\
& \wedge (userpmap_2 = pid_1 \wedge tlb_2 \neq entry_1 \rightarrow inactive) \\
& \wedge (userpmap_2 = pid_1 \wedge tlb_2 \neq entry_1 \rightarrow interrupted \vee (\textsc{Init}_{12} \leq pc_2 \leq \textsc{Init}_{13} \wedge pmap_2 = userpmap_2) \\
& \wedge (actionlock_1 \wedge \neg(\textsc{Resp}_4 < pc_1 \leq \textsc{Resp}_7) \wedge userpmap_1 = pid_2 \rightarrow plock_2) \\
& \wedge (actionlock_1 \wedge \neg(\textsc{Resp}_4 < pc_1 \leq \textsc{Resp}_7) \wedge userpmap_1 = pmap_2 \\
& \qquad\qquad\qquad \wedge \textsc{Init}_2 < pc_2 \leq \textsc{Init}_{14} \rightarrow \textsc{Init}_5 < pc_2 \leq \textsc{Init}_7 \wedge cpu_2 = pid_1 \\
& \wedge (actionlock_1 \wedge (\textsc{Init}_2 < pc_1 \leq \textsc{Init}_{14}) \rightarrow userpmap_1 \neq pmap_1))
\end{aligned}
$$

where
$$
\begin{aligned}
actneed &= actionneeded_2 \vee \textsc{Resp}_5 \leq pc_2 \leq \textsc{Resp}_6 \vee \textsc{Boot}_1 \leq pc_2 \leq \textsc{Boot}_2 \\
dirty &= (actionneeded_2 \wedge pc_2 \neq \textsc{Resp}_6) \vee (\textsc{Init}_{12} \leq pc_2 \leq \textsc{Init}_{13} \wedge userpmap_2 = pmap_2) \\
interrupted &= interrupt_2 \vee \textsc{Resp}_1 \leq pc_2 \leq \textsc{Resp}_8 \vee \textsc{Boot}_1 \leq pc_2 \leq \textsc{Boot}_2 \\
inactive &= \neg(\textsc{Main}_1 \leq pc_2 \leq \textsc{Main}_2)
\end{aligned}
$$

Figure 10. Invariant for the TLB shootdown algorithm for thread-modular reasoning at level 2. We omit the symmetric part where process 1 and 2 change their roles.

papers is to identify automatic techniques to solve the Horn constraints, paving the way to automatize k-thread-modular proofs for programs. Our work shines new light on these techniques from a proof-systems perspective; in particular, we prove that the number of quantifiers introduces a strict hierarchy for expressiveness but the method remains incomplete no matter how many quantifiers are used (and that the former continues to hold in the finite-state case but the latter does not).

While we have presented our proof system in the simple context of parameterized programs our proof system remains applicable when the program consists of a fixed number of threads (the setting of potentially different threads can be accommodated with a single thread template, through well-known encoding tricks). The size of the verification condition for a thread-modular proof at level k is $O(|P|^{k+1})$, that is, *polynomial* in the size of the template $|P|$ and exponential in the level k. In contrast, the size of the verification condition for the product construction on n threads is $O(|P|^n)$, exponential in the number of threads. Thus, even in the fixed-thread, non-parameterized setting, it is interesting to look for thread-modular proofs at higher levels as an alternative to introducing auxiliary variables.

We now compare our proof system with algorithms for parameterized verification for concurrent programs. While there are several verification tools for concurrent software based on practically useful heuristics [29, 38, 50], the theoretical power of these heuristics is often unclear. Moreover, many heuristics (such as auxiliary state introduction [24, 38]) seem orthogonal to k-thread modularity.

The method of *invisible invariants* [7, 48] for parameterized program constructs a candidate for an Ashcroft invariant by first computing the set of reachable states of the instance of the program with k threads, and then generalizing the concrete thread ids in the reachable states. The candidate (a universally quantified formula with k variables over thread ids) is then checked for inductiveness (using a syntactic "cutoff theorem"). However, the heuristics may not yield an Ashcroft invariant even if there is one (with the same number of quantifiers). Attempts to search for Ashcroft invariants satisfying a template [21] similarly suffer from the same problem (that is, if the method fails, one does not know whether there is no proof with k quantifiers or whether the heuristic did not find it). In contrast, our results provide a way to find Ashcroft invariants when they exist (modulo incompleteness of the Horn clause solver) or to prove they do not.

In conclusion, our results provide a unifying view of many similar results that have been used, implicitly or explicitly, in work on automatic verification for parameterized systems. The proof system of k-thread-modular proof rules is useful not only as a basis for automatic verification but also for proving systems that may presently be out of the scope of existing approaches to automatic verification.

References

[1] Martín Abadi and Leslie Lamport. Conjoining specifications. *Transactions on Programming Languages and Systems*, 17(3), 1995.

[2] P. A. Abdulla, K. Čerans, B. Jonsson, and Y.-K. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160(1–2), 2000.

[3] Parosh Aziz Abdulla, Yu-Fang Chen, Giorgio Delzanno, Frédéric Haziza, Chih-Duo Hong, and Ahmed Rezine. Constrained monotonic abstraction: A CEGAR for parameterized verification. In *CONCUR*, 2010.

[4] Parosh Aziz Abdulla, Frédéric Haziza, and Lukás Holík. All for the price of few. In *VMCAI*, 2013.

[5] Parosh Aziz Abdulla, Frédéric Haziza, and Lukás Holík. Parameterized verification through view abstraction. *STTT*, 18(5), 2016.

[6] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 2009.

[7] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jiazhao Xu, and Lenore D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, 2001.

[8] Edward A. Ashcroft. Proving assertions about parallel programs. *J. Comput. Syst. Sci.*, 10(1), 1975.

[9] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. *STTT*, 5(1), 2003.

[10] Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, 2008.

[11] Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, 2008.

[12] David L. Black. Personal communication. , 2016.

[13] David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Baron. Translation lookaside buffer consistency: A software approach. In *ASPLOS-III*, 1989.

[14] Ariel Cohen and Kedar S. Namjoshi. Local proofs for global safety properties. *Formal Methods in System Design*, 34(2), 2009.

[15] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[16] Patrick Cousot and Radhia Cousot. Reasoning about program invariance proof methods. Res. rep. CRIN-80-P050, Centre de Recherche en Informatique de Nancy (CRIN), Institut National Polytechnique de Lorraine, Nancy, France, July 1980.

[17] Patrick Cousot and Radhia Cousot. Invariance proof methods and analysis techniques for parallel programs. In *Automatic program construction techniques*. Macmillan, 1984.

[18] Alastair Donaldson, Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *CAV*, volume 6806, 2011.

[19] Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl. Counterexample-guided abstraction refinement for symmetric concurrent programs. *Formal Methods in System Design*, 41(1), 2012.

[20] Klaus Dräger, Andrey Kupriyanov, Bernd Finkbeiner, and Heike Wehrheim. SLAB: A certifying model checker for infinite-state concurrent systems. In *TACAS*, 2010.

[21] Michael Emmi, Rupak Majumdar, and Roman Manevich. Parameterized verification of transactional memories. In *PLDI*, 2010.

[22] Azadeh Farzan and Zachary Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *POPL*, 2012.

[23] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Inductive data flow graphs. In *POPL*, 2013.

[24] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proofs that count. In *POPL*, 2014.

[25] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proof spaces for unbounded parallelism. In *POPL*, 2015.

[26] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proving liveness of parameterized programs. In *LICS*, 2016.

[27] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Thread-modular verification for shared-memory programs. In *ESOP*, 2002.

[28] Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia. Modular verification of multithreaded programs. *Theoretical Computer Science*, 338(1-3), 2005.

[29] Silvio Ghilardi and Silvio Ranise. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *Logical Methods in Computer Science*, 6(4), 2010.

[30] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[31] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *CAV*, 2011.

[32] Arie Gurfinkel, Sharon Shoham, and Yuri Meshman. SMT-based verification of parameterized systems. In *FSE*, 2016.

[33] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *PLDI*, 2004.

[34] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *CAV*, 2003.

[35] Hossein Hojjat, Philipp Rümmer, Pavle Subotic, and Wang Yi. Horn clauses for communicating timed systems. In *Workshop on Horn Clauses for Verification and Synthesis*, 2014.

[36] Joxan Jaffar and Andrew E. Santosa. Recursive abstractions for parameterized systems. In *FM*, 2009.

[37] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and Systems*, 5(4), 1983.

[38] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Lost in abstraction: Monotonicity in multi-threaded programs. In *CONCUR*, 2014.

[39] Shuvendu K. Lahiri, Alexander Malkis, and Shaz Qadeer. Abstract threads. In *VMCAI*, 2010.

[40] Leslie Lamport. Proving the correctness of multiprocess programs. *Transactions on Software Engineering*, 3(2), 1977.

[41] R. Lipton. The reachability problem is exponential-space hard. Technical Report 62, Department of Computer Science, Yale University, 1976.

[42] Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Thread-modular verification is Cartesian abstract interpretation. In *ICTAC*, 2006.

[43] Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Precise thread-modular verification. In *SAS*, 2007.

[44] David Monniaux and Laure Gonnord. Cell morphing: From array programs to array-free horn clauses. In *SAS*, 2016.

[45] Kedar S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *VMCAI*, 2007.

[46] Leonor Prensa Nieto. Completeness of the Owicki-Gries system for parameterized parallel programs. In *IPDPS*, 2001.

[47] Susan S. Owicki. *Axiomatic proof techniques for parallel programs*. PhD thesis, Cornell University, 1975.

[48] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, 2001.

[49] Hartley Rogers, Jr. *Theory of recursive functions and effective computability*. MIT Press, 1987.

[50] Alejandro Sánchez and César Sánchez. Parametrized invariance for infinite state processes. *Acta Inf.*, 52(6), 2015.

[51] Alejandro Sánchez, Sriram Sankaranarayanan, César Sánchez, and Bor-Yuh Evan Chang. Invariant generation for parametrized systems using self-reflection. In *SAS*, 2012.

[52] Michal Segalov, Tal Lev-Ami, Roman Manevich, Ganesan Ramalingam, and Mooly Sagiv. Abstract transformers for thread correlation analysis. In *APLAS*, 2009.

[53] Natarajan Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR*, 2000.