Proof Spaces for Unbounded Parallelism

Azadeh Farzan Zachary Kincaid
University of Toronto

Andreas Podelski
University of Freiburg

Abstract

In this paper, we present a new approach to automatically verify multi-threaded programs which are executed by an unbounded number of threads running in parallel.

The starting point for our work is the problem of how we can leverage existing automated verification technology for sequential programs (abstract interpretation, Craig interpolation, constraint solving, etc.) for multi-threaded programs. Suppose that we are given a correctness proof for a trace of a program (or for some other program fragment). We observe that the proof can always be decomposed into a finite set of Hoare triples, and we ask *what can be proved from the finite set of Hoare triples using only simple combinatorial inference rules* (without access to a theorem prover and without the possibility to infer genuinely new Hoare triples)?

We introduce a proof system where one proves the correctness of a multi-threaded program by showing that for each trace of the program, *there exists* a correctness proof in the space of proofs that are derivable from a finite set of axioms using simple combinatorial inference rules. This proof system is complete with respect to the classical proof method of establishing an inductive invariant (which uses thread quantification and control predicates). Moreover, it is possible to algorithmically check whether a given set of axioms is sufficient to prove the correctness of a multi-threaded program, using ideas from well-structured transition systems.

1. Introduction

In this paper, we present a new approach to verifying multi-threaded programs which are executed by an unbounded number of concurrent threads. Many important systems and application programs belong to this category, e.g. filesystems, device drivers, web servers, and image processing applications.

We will start by demonstrating our approach on a simple example. Consider a program in which an unknown number of threads concurrently execute the code below. The goal is to verify that, if $\mathtt{g} \geq 1$ holds initially, then it will always hold (regardless of how many threads are executing).

```
global int g
local  int x
1: x := g;
2: g := g+x;
```

Consider the set of the Hoare triples (A) - (D) given below.

POPL'15, January 15 - 17 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01...$15.00.
http://dx.doi.org/10.1145/2676726.2677012

(A)	$\{\mathtt{g} \geq 1\}$	$\langle \mathtt{x := g} : 1\rangle$	$\{\mathtt{x}(1) \geq 1\}$
(B)	$\{\mathtt{g} \geq 1 \wedge \mathtt{x}(1) \geq 1\}$	$\langle \mathtt{g := g + x} : 1\rangle$	$\{\mathtt{g} \geq 1\}$
(C)	$\{\mathtt{g} \geq 1\}$	$\langle \mathtt{x := g} : 1\rangle$	$\{\mathtt{g} \geq 1\}$
(D)	$\{\mathtt{x}(1) \geq 1\}$	$\langle \mathtt{x := g} : 2\rangle$	$\{\mathtt{x}(1) \geq 1\}$

Here we use $\mathtt{x}(1)$ to refer to Thread 1's copy of the local variable x, and $\langle \mathtt{x := g} : 1\rangle$ to indicate the instruction `x := g` executed by Thread 1.

We will discuss how such Hoare triples can be generated automatically shortly. But for now, suppose that we have been given the above set of Hoare triples, and consider a deductive system in which these triples are taken as axioms, and the only rules of inference are *sequencing*, *symmetry*, and *conjunction*. These rules are easily illustrated with concrete examples:

- *Sequencing* composes two Hoare triples sequentially. For example, sequencing (A) and (D) yields

  $$(\text{A} \circ \text{D}) \quad \{\mathtt{g} \geq 1\} \quad \langle \mathtt{x := g} : 1\rangle\langle \mathtt{x := g} : 2\rangle \quad \{\mathtt{x}(1) \geq 1\}$$

- *Symmetry* permutes thread identifiers. For example, renaming (A) and (C) (mapping $1 \mapsto 2$) yields

  $$\begin{array}{lll} (\text{A'}) & \{\mathtt{g} \geq 1\} \quad \langle \mathtt{x := g} : 2\rangle & \{\mathtt{x}(2) \geq 1\} \\ (\text{C'}) & \{\mathtt{g} \geq 1\} \quad \langle \mathtt{x := g} : 2\rangle & \{\mathtt{g} \geq 1\} \end{array}$$

  and, renaming (D) (mapping $1 \mapsto 2$ and $2 \mapsto 1$) yields

  $$(\text{D'}) \quad \{\mathtt{x}(2) \geq 1\} \quad \langle \mathtt{x := g} : 1\rangle \quad \{\mathtt{x}(2) \geq 1\}$$

- *Conjunction* composes two Hoare triples by conjoining pre- and postconditions. For example, conjoining (A') and (C') yields

  $$(\text{A'} \wedge \text{C'}) \quad \{\mathtt{g} \geq 1\} \quad \langle \mathtt{x := g} : 2\rangle \quad \{\mathtt{g} \geq 1 \wedge \mathtt{x}(2) \geq 1\}$$

  and conjoining (A) and (D') yields $(\text{A} \wedge \text{D'})$

  $$\{\mathtt{g} \geq 1 \wedge \mathtt{x}(2) \geq 1\} \quad \langle \mathtt{x := g} : 1\rangle \quad \{\mathtt{x}(1) \geq 1 \wedge \mathtt{x}(2) \geq 1\}$$

Naturally, the deductive system may apply inference rules to deduced Hoare triples as well: for example, by sequencing $(\text{A'} \wedge \text{C'})$ and $(\text{A} \wedge \text{D'})$, we get the Hoare triple

$$\{\mathtt{g} \geq 1\} \quad \langle \mathtt{x := g} : 2\rangle\langle \mathtt{x := g} : 1\rangle \quad \{\mathtt{x}(1) \geq 1 \wedge \mathtt{x}(2) \geq 1\}$$

A *proof space* is a set of valid Hoare triples which is closed under sequencing, symmetry, and conjunction (that is, it is a *theory* of this deductive system). Any finite set of valid Hoare triples generates an infinite proof space by considering those triples to be axioms and taking their closure under deduction; we call such a finite set of Hoare triples a *basis* for the generated proof space.

One key insight in this paper is that, although it may be challenging to formalize a correctness argument for a multi-threaded program, any given *trace* (sequence of program instructions) of the program is just a simple (straight-line) sequential program which

can be easily proved correct. In fact, just these three simple inference rules (sequencing, symmetry, and conjunction) are sufficient to prove the correctness of *any* trace of the example program. That is, for any trace τ of the program, $\{\mathtt{g} \geq 1\} \quad \tau \quad \{\mathtt{g} \geq 1\}$ belongs to the proof space generated by (A) – (D), regardless of *which* or *how many* threads execute in τ. Crucially (and *due* to the simplicity of the inference rules), this fact can be checked completely automatically, by leveraging techniques developed in the context of well-structured transition systems [2, 17, 18].

Now let us turn to the problem of how Hoare triples like (A) – (D) can be generated automatically. An important feature of proof spaces is that they are generated from a very simple set of Hoare triples, of the sort one could expect to generate using standard technology for sequential verification. To appreciate the simplicity of these triples, consider the following inductive invariant for the program, the classical notion of correctness proof for multi-threaded programs:

$$\mathtt{g} \geq 1 \wedge (\forall i. \mathsf{loc}(i) = 2 \Rightarrow \mathtt{x}(i) \geq 1)$$

(which indicates that g is at least 1, and all threads i at line 2 of the program have $\mathtt{x}(i)$ at least 1). This is a simple invariant for this program, but it is one which we cannot rely on a sequential verifier to find because it makes use of features which are not encountered on sequential programs: thread quantification ($\forall i$), and control predicates ($\mathsf{loc}(i) = 2$). In contrast, the triples (A) – (D) are of the form one might expect to generate using a sequential verifier. Thus, proof spaces are a solution to the problem of *how to use automated proof technology for sequential programs to prove the correctness of programs with unboundedly many threads*.

One possible algorithm (discussed further in Section 7) for automating proof spaces is to build a basis for a proof space iteratively in a manner analogous to a counter-example guided abstraction refinement (CEGAR) loop. While this algorithm is not the focus of this paper, it is important to recognize the context of and motivation behind proof spaces. The algorithm operates as follows: first choose a trace for which there is no correctness theorem in the proof space. For example, suppose that we start with the trace

$$\langle \mathtt{x} := \mathtt{g} : 1\rangle\langle \mathtt{x} := \mathtt{g} : 2\rangle\langle \mathtt{g} := \mathtt{g} + \mathtt{x} : 1\rangle$$

This trace can be viewed as a sequential program, and we can use standard technology (e.g., abstract interpretation, constraint solving, Craig interpolation) to construct a proof for it (for example, the one on the right). By extracting the atomic Hoare triples along this sequence, we can arrive at the triples (A), (B), and (D), which we add to our basis. We may then extract a new trace which cannot be proved using just (A), (B), and (D), and restart the loop. The loop terminates when (and if) (1) it finds a feasible counter-example or (2) all traces of the program have correctness theorems in the proof space.

$\{\mathtt{g} \geq 1\}$
$\langle \mathtt{x} := \mathtt{g} : 1\rangle$
$\{\mathtt{x}(1) \geq 1\}$
$\langle \mathtt{x} := \mathtt{g} : 2\rangle$
$\{\mathtt{x}(1) \geq 1\}$
$\langle \mathtt{g} := \mathtt{g} + \mathtt{x} : 1\rangle$
$\{\mathtt{g} \geq 1\}$

The key contribution of this paper is the notion of *proof spaces*, a proof system which can be used to exploit sequential verification technology for proving the correctness of multi-threaded programs with unboundedly many threads. The merits of proof spaces are as follows:

- Rather than enriching the language of assertions (introducing thread quantification and control predicates) and employing more powerful symbolic reasoning, proof spaces use simple assertions which can be combined using combinatorial reasoning (i.e., without a theorem prover).
- We show that the combinatorial reasoning involved in checking whether all traces of a program have a correctness theorem in a proof space can be automated, and give an algorithm which leverages ideas from well-structured transition systems [18]. We show that this proof space checking problem is undecidable in general, but our algorithm is a decision procedure for an interesting subclass of proof spaces.
- Despite the apparent weakness of our three inference rules and the restricted language of assertions, we show that proof spaces are complete relative to a variant of Ashcroft's classical proof system for multi-threaded programs (which employs universal thread quantification and control predicates).

2. Motivating example

Consider a simplified implementation of a thread pool, where there are arbitrarily many threads each executing the code that appears in Figure 1. The global variable `tasks` holds an array of tasks of size `len`, the global variable `next` stores the index of the next available task, and the global variable `m` is a lock which protects access to `next`. We denote the threads of the program by $T_1, T_2, \ldots$. Each thread $T_\mathtt{i}$ has two local variables `c(i)` and `end(i)` which represent the current and last task in the block of tasks acquired by $T_\mathtt{i}$. Each thread operates by simply acquiring a block of 10 consecutive tasks (lines 1-8) and then performing the tasks in its acquired block in sequence (lines 9-13). The `else` branch (line 6) ensures that once the end of the array is reached, all remaining threads that attempt to acquire a block of tasks acquire an empty block (`c = end`).

The details of the tasks are omitted for simplicity, and we maintain only enough information to assert a desired property: that no two threads are ever assigned the same task. This property is encoded by using the `tasks` array to represent the status of each task (0 = started, 1 = finished); a thread can fail the assertion if, between finishing some task and executing the assertion, some other thread starts that same task.

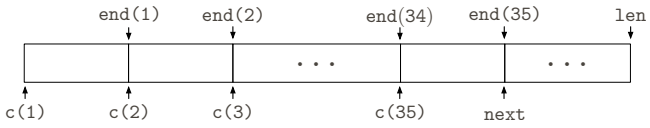
```
global int : len; // total number of tasks
global int array(len) : tasks; // array of tasks
global int : next; // position of next available task block
global lock : m; // lock protecting next

thread T:
   local int : c;    // position of current task
   local int : end; // position of last task in acquired block
   // acquire block of tasks
1  lock(m);
2     c := next;
3     next := next + 10;
4     if (next <= len):
5        end := next;
6     else:
7        end := len;
8  unlock(m);
   // perform block of tasks
9  while (c < end):
10    tasks[c] := 0; // mark task c as started
      ...            // work on the task c
11    tasks[c] := 1; // mark task c as finished
12    assert(tasks[c] == 1); // no other thread has started task c
13    c := c + 1;
```

Figure 1. Thread pooling example, adapted from [36].

The sketch below illustrates a snapshot of the `tasks` array (and other instantiated local and global variables) in the thread pool program, where threads 1 to 35 have all finished acquiring their block of tasks (but have not yet started to complete them), and thread 36 has not started the process yet.

Locking	$\{true\}$ $\langle$`lock(m)` : 1$\rangle$ $\{m = 1\}$	Initialization	$\{true\}$ $\langle$`c := next` : 1$\rangle$ $\{c(1) \leq next\}$	$\{c(1) \leq next\}$ $\langle$`next := next + 10` : 1$\rangle$ $\{c(1) < next\}$	$\{true\}$ $\langle$`end := next` : 1$\rangle$ $\{end(1) \leq next\}$
	$\{m = 1\}$ $\langle$`lock(m)` : 1$\rangle$ $\{false\}$		$\{end(1) \leq next\}$ $\langle$`c := next` : 2$\rangle$ $\{end(1) \leq c(2)\}$	$\{true\}$ $\langle$`assume(next > len)` : 1$\rangle$ $\{len \leq next\}$	$\{len \leq next\}$ $\langle$`end := len` : 1$\rangle$ $\{end(1) \leq next\}$
Loop	$\{true\}$ $\langle$`assume(c < end)` : 1$\rangle$ $\{c(1) < end(1)\}$	$\{true\}$ $\langle$`tasks[c] := 1` : 1$\rangle$ $\{tasks[c(1)] = 1\}$	$\{tasks[c(1)] = 1\}$ $\langle$`assume(tasks[c] != 1)` : 1$\rangle$ $\{false\}$	$\{end(1) \leq c(2)\}$ $\langle$`c := c + 1` : 2$\rangle$ $\{end(1) \leq c(2)\}$	
	$\{tasks[c(1)] = 1 \wedge c(1) < end(1) \leq c(2)\}$ $\langle$`tasks[c] := 0` : 2$\rangle$ $\{tasks[c(1)] = 1\}$		$\{tasks[c(1)] = 1 \wedge c(2) < end(2) \leq c(1)\}$ $\langle$`tasks[c] := 0` : 2$\rangle$ $\{tasks[c(1)] = 1\}$		

Figure 2. The basis of a proof space for the thread pooling example. The variable `c(1)` denotes the copy of the local variable `c` in Thread 1 and the command $\langle$`c := next` : 1$\rangle$ denotes the instance of the command `c := next` in Thread 1, etc. Trivial Hoare triples of the type $\{\varphi\}$ $\langle\sigma$: `i`$\rangle$ $\{\varphi\}$ where the set of variables that $\langle\sigma$: `i`$\rangle$ modifies is disjoint from the set of variables in φ, have been omitted for brevity.



In order to show that two threads, say Threads 1 and 2, cannot be assigned the same task, we may argue that the intervals assigned to them `[c(1), end(1))` and `[c(2), end(2))` are disjoint by proving that `end(1)` is less than or equal to `c(2)`.

Proof space for thread pooling Figure 2 illustrates a finite set of Hoare triples (a basis) that generates a proof space for the thread pooling program. In the figure, the Hoare triples are categorized into three groups (*Locking*, *Initialization*, and *Loop*). We omit Hoare triples that represent trivial invariance axioms of the form $\{\varphi\}$ $\langle\sigma$: `i`$\rangle$ $\{\varphi\}$ where the command $\langle\sigma$: `i`$\rangle$ does not write to a variable that appears in φ; for example,

$$\{c(2) \leq next\}\ \langle \texttt{c := next} : 1\rangle\ \{c(2) \leq next\}\ .$$

Note that the set of triples in Figure 2 use *mixed assertions* (in the terminology in [12]), which relate the values of local variables to global variables (e.g., $c(1) \leq$ `next`), as well as *inter-thread assertions*, which relate the local variables of different threads (e.g., $end(1) \leq c(2)$). As discussed in [25], both mixed assertions and inter-thread assertions raise an issue when one attempts to use predicate abstraction techniques. Inter-thread assertions cannot be used in classical compositional proof systems (e.g. thread-modular proofs). In these proof systems, inter-thread relationships can only be accommodated through auxiliary variables [29, 35].

To represent the program in a simple formal model, we encode each conditional branch as a nondeterministic branch between `assume` commands (one for the condition and one for its negation), and we encode `assert(e)` through a command `assume(!e)` (for the negation of the expression `e`) which leads to a new *error location*. We thus encode the correctness of the program (the validity of the assert statement) through the non-reachability of the error location by any thread. An *error trace* is an interleaved sequence of commands of any number of threads which leads *some* thread to the error location (e.g., in Figure 3, the sequence in (a) followed by the sequence in (b)). Thus, we can express the correctness of the program by the validity of the Hoare triple $\{true\}\ \tau\ \{false\}$ for every error trace τ. Given this Hoare triple as the specification of correctness of a trace, we have that *a trace is correct if and only if it is infeasible*.

Let us now demonstrate how the proof space is used to argue for the correctness of the program. We must show that for every error trace τ *there exists* a derivation of $\{true\}\ \tau\ \{false\}$ which can be constructed from the triples in Figure 2 using only the combinatorial inference rules of symmetry, conjunction, and sequencing.

Let us first consider the pair of Hoare triples in the *Locking* group. We treat the `lock(m)` command as the atomic sequence `{assume(m = 0); m := 1}`, and `unlock(m)` command as the assignment `m := 0`. Intuitively, the locking Hoare triples encapsulate the reasoning required to prove that the lock `m` provides mutually exclusive access to the variable `next`. Any trace that violates the locking semantics can be proved infeasible using the Hoare triples in the Locking group along with the sequencing and symmetry rules. To see why, consider that any such trace can be decomposed as

$$\tau_1 \cdot \langle \texttt{lock(m)} : \texttt{i}\rangle \cdot \tau_2 \cdot \langle \texttt{lock(m)} : \texttt{j}\rangle \cdot \tau_3$$

such that the following Hoare triples are valid:

$$\begin{array}{ccc} \{true\} & \tau_1 & \{true\} \\ \{true\} & \langle \texttt{lock(m)} : \texttt{i}\rangle & \{m = 1\} \\ \{m = 1\} & \tau_2 & \{m = 1\} \\ \{m = 1\} & \langle \texttt{lock(m)} : \texttt{j}\rangle & \{false\} \\ \{false\} & \tau_3 & \{false\} \end{array}$$

This decomposition exploits the fact that any trace which violates the semantics of locking has a shortest prefix which contains a violation: that is, we may assume that τ_2 contains no `unlock(m)` commands because, if it did, there would be a shorter prefix in

which two threads simultaneously hold the lock `m`. Having justified their semantic validity, we may now consider how to derive these triples using symmetry and sequencing from the Locking axioms in Figure 2. The two triples above concerning the `lock(m)` command are inferred from the triples in the Locking group by renaming thread 1 (i.e. using the symmetry rule) to `i` and `j`, respectively. The rest of the triples come from the simple invariance Hoare triples (not depicted in Figure 2, but mentioned in the caption), that allow us to infer $\{m = 1\}\ \ \langle\sigma : \mathtt{i}\rangle\ \ \{m = 1\}$ for any command σ except `unlock(m)`, and $\{\mathit{true}\}\ \ \langle\sigma : \mathtt{i}\rangle\ \ \{\mathit{true}\}$ and $\{\mathit{false}\}\ \ \langle\sigma : \mathtt{i}\rangle\ \ \{\mathit{false}\}$ for every command σ.

We now turn our attention to the error traces which *do* respect locking semantics, and show that they are infeasible. The six Hoare triples in the *Initialization* section of Figure 2 are sufficient to prove that after two threads (say 2 and 9) acquire their block of tasks, those blocks do not overlap (i.e., we have either $\mathtt{end}(9) \leq \mathtt{c}(2)$ or $\mathtt{end}(2) \leq \mathtt{c}(9)$, depending on the order in which the threads acquire their tasks). An example of such a trace (which can be proved using just sequencing and symmetry operations) appears in Figure 3(a). We encourage the reader to show (using conjunction) that if we extend the trace in Figure 3(a) by the initialization sequence of a third thread (say, Thread 5) to obtain a trace τ, then

$$\{\mathit{true}\}\ \ \tau\ \ \{\mathtt{end}(2) \leq \mathtt{c}(9) \wedge \mathtt{end}(2) \leq \mathtt{c}(5) \wedge \mathtt{end}(9) \leq \mathtt{c}(5)\}$$

belongs to the proof space as well. Following similar proof combination steps, one can see that the argument can be adapted to traces with any number of threads.

Finally, the triples in the *Loop* section can be used to show two things. First, the "non-overlapping" property established in the initialization section is preserved by the loop: for example, if $\mathtt{end}(2) \leq \mathtt{c}(9)$ holds at the beginning of the loop, then if thread 2 or thread 9 (or any other thread) executes the loop, $\mathtt{end}(2) \leq \mathtt{c}(9)$ continues to hold. Second, (assuming the non-overlapping condition holds), if some task has been completed then it remains completed: for example, if `tasks[c(2)] = 1`, then when thread 9 starts task `c(9)` (i.e. assigns 0 to `tasks[c(9)]`), it cannot overwrite the value of the array cell `tasks[c(2)]` (this is ensured by the precondition $\mathtt{c}(2) < \mathtt{end}(2) \leq \mathtt{c}(9)$, which implies $\mathtt{c}(2) \neq \mathtt{c}(9)$). Figure 3(b) gives an example proof which can be derived using the *Loop* section. The sequential composition of traces in Figure 3(a) and Figure 3(b) forms an error trace (i.e. one that leads to the error location), and sequencing their proofs yields a proof that the error trace is infeasible.

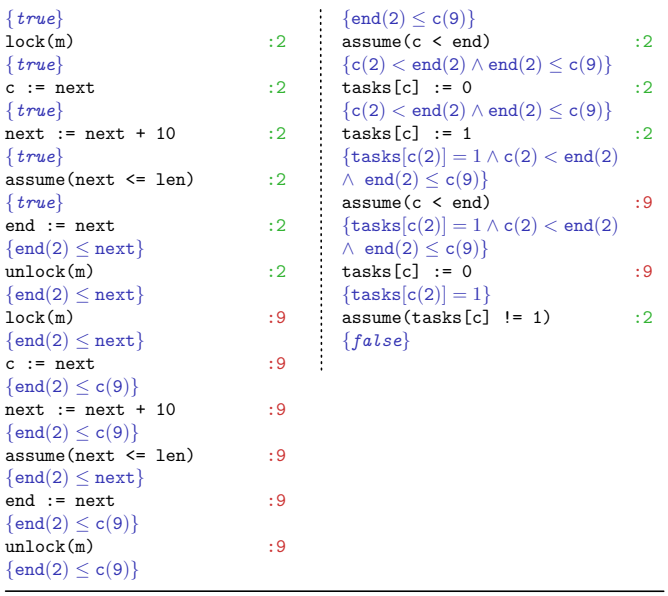
We hope that it is now intuitively clear (or at least plausible) that, given any particular error trace τ, it is possible to derive $\{\mathit{true}\}\ \ \tau\ \ \{\mathit{false}\}$ using the symmetry, sequencing, and conjunction rules starting from the axioms given in Figure 2. The question is: how can one be assured that all (infinitely many) error traces can be proved infeasible in this way? In Section 6 we will show how to formalize such an argument, and moreover, give a procedure for checking that it holds.

3. Preliminaries

In this section, we will define our program model and introduce some technical definitions.

We fix a set of global variables GV and a set of local variables LV. Intuitively, these are the variables that can appear in the program text. We will often make use of the set $\mathsf{LV} \times \mathbb{N}$ of *indexed* local variables, and denote a pair $\langle x, \mathtt{i}\rangle \in \mathsf{LV} \times \mathbb{N}$ by $x(\mathtt{i})$. Intuitively, such an indexed local variable $x(\mathtt{i})$ refers to thread `i`'s copy of the local variable x.

For simplicity, we will assume that program variables take integer values, and the program is expressible in the theory of linear integer arithmetic. This assumption is made only to simplify the presentation by making it more concrete; our approach and results



(a) Initialization example (b) Loop example

Figure 3. Example traces

do not depend on linear arithmetic in any essential way.

Given a set of variable symbols V, we use $\mathsf{Term}(V)$ to denote the set of linear terms with variables drawn from V. Similarly, $\mathsf{Formula}(V)$ denotes the set of linear arithmetic formulas with variables drawn from V. We here use V as a parameter which we will instantiate according to the context. For example, $\mathsf{Term}(\mathsf{GV} \cup \mathsf{LV})$ denotes the set of terms which may appear in the text of a program (which refers to the thread template), and $\mathsf{Formula}(\mathsf{GV} \cup (\mathsf{LV} \times \mathbb{N}))$ denotes the set of formulas which may appear in a pre- or postcondition (which refers to concrete threads).

We use the notation $[x \mapsto t]$ to denote a substitution which replaces the variable x with the term t and generalize the notation to parallel substitutions in the standard way. The application of a substitution ρ to a formula φ (or a term t) is denoted $\varphi[\rho]$ (or $t[\rho]$). Two classes of substitutions of particular interest for our technical development are *instantiations* and *permutations*.

- Given an index $\mathtt{i} \in \mathbb{N}$, we use $\iota_{\mathtt{i}}$ to denote the instantiation substitution which replaces each local variable $x \in \mathsf{LV}$ with an indexed local variable $x(\mathtt{i})$. For example, if we take `i` to be 3 and `c` is a local variable and `next` is a global, we have
$$(\mathtt{c} \leq \mathtt{next})[\iota_3] = \mathtt{c}(3) \leq \mathtt{next}$$
- Each permutation $\pi : \mathbb{N} \to \mathbb{N}$ defines a substitution which replaces each indexed local variable $x(\mathtt{i})$ with $x(\pi(\mathtt{i}))$. We abuse notation by identifying each permutation with the substitution it defines. For example if π is a permutation which maps $3 \leftrightarrow 1$, then
$$(\mathtt{c}(3) \leq \mathtt{next})[\pi] = \mathtt{c}(1) \leq \mathtt{next}$$

We identify a program P with a control flow graph for a single *thread template* which is executed by an unbounded number of threads. (The extension to more than one thread template is straightforward.) A control flow graph is a directed, labeled graph $P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, \ell_{\mathsf{err}}, \mathsf{src}, \mathsf{tgt}\rangle$, where Loc is a set of program locations, Σ is a set of program commands, ℓ_{init} is a designated initial location, ℓ_{err} is a designated error location, and $\mathsf{src}, \mathsf{tgt} : \Sigma \to \mathsf{Loc}$ are functions mapping each program command to its source and target location. Note that our definition of a control flow graph (using

src and tgt) implies that we distinguish between different occurrences of a program instruction.

A program command $\sigma \in \Sigma$ is of one of two forms:

- an *assignment* of the form $x := t$ where $x \in \mathsf{GV} \cup \mathsf{LV}$ and $t \in \mathsf{Term}(\mathsf{GV} \cup \mathsf{LV})$, or
- an *assumption* of the form $\texttt{assume}(\varphi)$ where $\varphi \in \mathsf{Formula}(\mathsf{GV} \cup \mathsf{LV})$.

We will denote a pair consisting of a program command $\sigma \in \Sigma$ and an index $\mathtt{i} \in \mathbb{N}$ as the *indexed* command $\langle \sigma : \mathtt{i} \rangle$. Intuitively, such an indexed command $\langle \sigma : \mathtt{i} \rangle$ refers to thread $\mathtt{i}$'s instance of the program command σ, or: the index $\mathtt{i}$ indicates the identifier of the thread which executes σ.

A *trace*

$$\tau = \langle \sigma_1 : \mathtt{i}_1 \rangle \langle \sigma_2 : \mathtt{i}_2 \rangle \cdots \langle \sigma_n : \mathtt{i}_n \rangle$$

is a sequence of indexed commands.

A *Hoare triple* is a triple

$$\{\varphi\} \ \ \tau \ \ \{\psi\}$$

where τ is a trace and φ and ψ are formulas with global variables and indexed local variables, i.e., $\varphi, \psi \in \mathsf{Formula}(\mathsf{GV} \cup (\mathsf{LV} \times \mathbb{N}))$.

The validity of Hoare triples for traces is defined as one would expect. Intuitively, if the command reads or writes the local variable x, then the indexed command $\langle \sigma : \mathtt{i} \rangle$ (i.e., the command σ executed by thread $\mathtt{i}$) reads or writes thread $\mathtt{i}$'s copy of the local variable x. To express this formally, we use the instantiation substitution $\iota_\mathtt{i}$ introduced above. In particular, if x is a local variable, $x[\iota_\mathtt{i}]$ is the indexed local variable $x(\mathtt{i})$ (and if x is a global variable, $x[\iota_\mathtt{i}]$ is simply x).

For a trace of length 1 (i.e., for an indexed command), we have:

- $\{\varphi\} \ \ \langle (x := t) : \mathtt{i} \rangle \ \ \{\psi\}$ is valid if
$$\varphi \models \psi[x[\iota_\mathtt{i}] \mapsto t[\iota_\mathtt{i}]]$$
- $\{\varphi\} \ \ \langle (\texttt{assume}(\theta)) : \mathtt{i} \rangle \ \ \{\psi\}$ is valid if
$$\varphi \wedge \theta[\iota_\mathtt{i}] \models \psi$$

where $\models$ denotes entailment modulo the theory of integer linear arithmetic. For a trace of the form $\tau.\langle \sigma : \mathtt{i} \rangle$, the triple $\{\varphi\} \ \ \tau.\langle \sigma : \mathtt{i} \rangle \ \ \{\psi\}$ is valid if there exists some formula ψ' such that $\{\varphi\} \ \ \tau \ \ \{\psi'\}$ and $\{\psi'\} \ \ \langle \sigma : \mathtt{i} \rangle \ \ \{\psi\}$ are valid.

We call a trace τ *infeasible* if the Hoare triple $\{\mathit{true}\} \ \ \tau \ \ \{\mathit{false}\}$ is valid. Intuitively, τ is infeasible if it does not correspond to any execution.

Given a program P (identified by the control flow graph of a single thread template, i.e., a graph with edges labeled by commands $\sigma \in \Sigma$ and with the initial location ℓ_{init} and the error location ℓ_{err}), a trace τ is an *error trace* of P if

- for each index $\mathtt{i} \in \mathbb{N}$, the projection of τ onto the commands of thread $\mathtt{i}$ corresponds to a path through P starting at ℓ_{init} (i.e., each thread $\mathtt{i}$ starts at the initial location), and
- there is (at least) one index $\mathtt{j} \in \mathbb{N}$ such that the projection of τ onto the commands of thread $\mathtt{j}$ corresponds to a path through P ending at ℓ_{err} (i.e., some thread $\mathtt{j}$ ends at the error location).

We say that a program P is *correct* if every error trace of P is infeasible (i.e., there is no execution of P wherein some thread reaches the error location ℓ_{err}).

The notion of correctness be used to encode many correctness properties including thread-state reachability (and thus the safety of `assert` commands) and mutual exclusion. The encoding is possibly done by introducing monitors (ghost instructions on ghost variables), much in the same way that safety properties can be reduced to non-reachability in sequential programs.

4. Proof spaces

We will now introduce *proof spaces*, the central technical idea of this paper. We begin by formalizing the inference rules of SEQUENCING, SYMMETRY, and CONJUNCTION.

The *sequencing rule* is a modification of the familiar one from Hoare logic. We omit the rule of consequence from Hoare logic, and instead incorporate consequence in our sequencing rule: that is, we allow triples $\{\varphi_0\} \ \ \tau_0 \ \ \{\varphi_1\}$ and $\{\varphi_1'\} \ \ \tau_1 \ \ \{\varphi_2\}$ to be composed when φ_1 and φ_1' do not exactly match, but φ_1 entails φ_1'. However, a design goal of proof spaces is that the inference rules should be purely combinatorial (and thus, should not require access to a theorem prover to discharge the entailment). Towards this end, we define a *combinatorial entailment* relation $\Vdash$ on formulas: identifying a conjunction with the set of its conjuncts, we have $\varphi \Vdash \psi$ if φ is a superset of ψ. Formally,

$$\varphi_1 \wedge ... \wedge \varphi_n \Vdash \psi_1 \wedge ... \wedge \psi_m \ \text{ if } \ \{\varphi_1, ..., \varphi_n\} \supseteq \{\psi_1, ..., \psi_m\}$$

The sequencing rule is formalized as follows:

SEQUENCING
$$\frac{\{\varphi_0\} \ \ \tau_0 \ \ \{\varphi_1\} \qquad \varphi_1 \Vdash \varphi_1' \qquad \{\varphi_1'\} \ \ \tau_1 \ \ \{\varphi_2\}}{\{\varphi_0\} \ \ \tau_0 ; \tau_1 \ \ \{\varphi_2\}}$$

The symmetry rule exploits the fact that thread identifiers are interchangeable in our program model, and therefore uniformly permuting thread identifiers in a valid Hoare triple yields another valid Hoare triple:

SYMMETRY
$$\frac{\{\varphi\} \ \ \langle \sigma_1 : \mathtt{i}_1 \rangle \cdots \langle \sigma_n : \mathtt{i}_n \rangle \ \ \{\psi\}}{\{\varphi[\pi]\} \ \ \langle \sigma_1 : \pi(\mathtt{i}_1) \rangle \cdots \langle \sigma_n : \pi(\mathtt{i}_n) \rangle \ \ \{\psi[\pi]\}} \quad \pi : \mathbb{N} \to \mathbb{N} \text{ is a permutation}$$

Lastly, the rule of conjunction allows one to combine two theorems about the same trace by conjoining preconditions and postconditions:

CONJUNCTION
$$\frac{\{\varphi_1\} \ \ \tau \ \ \{\psi_1\} \qquad \{\varphi_2\} \ \ \tau \ \ \{\psi_2\}}{\{\varphi_1 \wedge \varphi_2\} \ \ \tau \ \ \{\psi_1 \wedge \psi_2\}}$$

Next, we formalize our notion of a proof space:

Definition 4.1 (Proof space) A *proof space* $\mathscr{H}$ is a set of valid Hoare triples which is closed under SEQUENCING, SYMMETRY, and CONJUNCTION. ⌟

Our definition restricts proof spaces to contain only valid Hoare triples for the sake of convenience. Clearly, the SEQUENCING, SYMMETRY, and CONJUNCTION rules preserve validity.

The strategy behind using proof spaces as correctness proofs can be summarized in the following proof rule: *If we can find a proof space $\mathscr{H}$ such that for every error trace τ of the program P, we have that the Hoare triple $\{\mathit{true}\} \ \ \tau \ \ \{\mathit{false}\}$ belongs to $\mathscr{H}$, then P is correct.*

Our main interest in proof spaces is using the above proof rule as a foundation for algorithmic verification of concurrent programs. Towards this end, we augment the definition of proof spaces with additional conditions that make them easier to manipulate algorithmically. We call the proof spaces satisfying these conditions *finitely generated proof spaces*. First, we define *basic Hoare triples*, which are the "generators" of such proof spaces.

Definition 4.2 (Basic Hoare triple) A *basic Hoare triple* is a valid Hoare triple of the form

$$\{\varphi\} \ \ \langle \sigma : \mathtt{i} \rangle \ \ \{\psi\}$$

where

1. the postcondition ψ cannot be constructed by conjoining two other formulas, and

2. for each $\mathtt{j} \in \mathbb{N}$, if an indexed local variable of the form $x(\mathtt{j})$ appears in the precondition φ, then either $\mathtt{j}$ is equal to the index $\mathtt{i}$ of the command, or $x(\mathtt{j})$ or some other indexed local variable $y(\mathtt{j})$ with the same index $\mathtt{j}$ must appear in the postcondition ψ ("the precondition mentions only relevant threads"). ⌟

The asymmetry between the precondition and the postcondition in Condition 1 is justified by the rule of consequence. Given a (non-basic) Hoare triple

$$\{\varphi_0 \wedge \varphi_1\} \ \langle \sigma : \mathtt{i} \rangle \ \{\psi_1 \wedge \psi_2\},$$

we may "split the postcondition" to arrive at two valid triples

$$\{\varphi_0 \wedge \varphi_1\} \ \langle \sigma : \mathtt{i} \rangle \ \{\psi_1\}$$
$$\{\varphi_0 \wedge \varphi_1\} \ \langle \sigma : \mathtt{i} \rangle \ \{\psi_2\}$$

from which the original triple may be derived via CONJUNCTION. As a result, the postcondition restriction on basic Hoare triples does not lose significant generality. Since preconditions cannot be split in this way, Definition 4.2 is asymmetric.

Definition 4.3 (Finitely generated) We say that a proof space $\mathscr{H}$ is *finitely generated* if there exists a finite set of basic Hoare triples H such that $\mathscr{H}$ is the smallest proof space which contains H. In this situation, we call H a *basis* for $\mathscr{H}$. ⌟

Proof spaces give a new proof system for verifying safety properties for multi-threaded programs. The remainder of this paper studies some of the questions which arise from introducing such a proof system. In the next section, we consider the question of the expressive power of proof spaces (that is, what can be proved using proof spaces). In Sections 6 and 7, we discuss how proof spaces may be used in the context of algorithmic verification.

5. Completeness

The method of using global inductive invariants to prove correctness of concurrent programs dates back to the seminal work of Ashcroft [4]. Ashcroft's method originally applied to programs with finitely many threads, but can be adapted to the infinite case by admitting into the language of assertions universal quantifiers over threads. In this section, we prove the completeness of proof spaces relative to Ashcroft's method. This establishes that the combinatorial operations of proof spaces are adequate for representing Ashcroft proofs, even though Ashcroft assertions make use of features not available to proof space assertions (namely, universal quantifiers over threads and control assertions).

We start by formalizing Ashcroft proofs. We can treat each local variable as an uninterpreted function symbol of type $\mathbb{N} \to \mathbb{Z}$ (i.e, the interpretation of $x \in \mathsf{LV}$ is a function which maps each thread identifier $\mathtt{i} \in \mathbb{N}$ to the value of thread $\mathtt{i}$'s copy of x).

We let TV be a set of *thread variables* which are variables whose values range over the set of thread identifiers $\mathbb{N}$. We typically use $i, j, i_1, \ldots$ to refer to thread variables (note that we use $\mathtt{i}, \mathtt{j}, \mathtt{i}_1, \ldots$ to refer to thread identifiers, i.e., elements of $\mathbb{N}$). We use $\mathsf{LV}[\mathsf{TV}]$ to denote the set of terms $x(i)$ where $x \in \mathsf{LV}$ and $i \in \mathsf{TV}$. A *data assertion* is a linear arithmetic formula ψ built up from global variables and terms of the form $x(i)$ where $x \in \mathsf{LV}$ and $i \in \mathsf{TV}$, i.e., $\psi \in \mathsf{Formula}(\mathsf{GV} \cup \mathsf{LV}[\mathsf{TV}])$.

A *global assertion* is defined to be a sentence of the form

$$\Phi_{\mathsf{inv}} = \forall i_1, ..., i_n . \varphi_{\mathsf{inv}}(i_1, ..., i_n)$$

where φ_{inv} is a formula built up from Boolean combinations of

- *data assertions*,
- control assertions $\mathsf{loc}(i) = \ell$ (with $i \in \mathsf{TV}$ and $\ell \in \mathsf{Loc}$), and
- thread equalities $i = j$ and disequalities $i \neq j$ (with $i, j \in \mathsf{TV}$).

For any program command $\sigma \in \Sigma$, we use $[\![\sigma]\!]$ to denote the transition formula which represents *some* thread executing σ. This is a formula over an extended vocabulary which includes a primed copy of each of the global and local variable symbols (interpreted as the post-state values of those variables). This formula is of the form

$$\exists i . \psi^1(i) \wedge (\forall j . \psi^2(i, j))$$

where (intuitively) ψ^1 describes the state change for the thread which executes σ (thread i), and ψ^2 describes the state change for all other threads (thread j). For example, if σ is an assignment $x := x + g$ where x is local, g is global, and $\ell = \mathsf{src}(\sigma)$ and $\ell' = \mathsf{tgt}(\sigma)$, then:

$$[\![\sigma]\!] \equiv \exists i . \Big(\mathsf{loc}(i) = \ell \wedge \mathsf{loc}'(i) = \ell' \wedge x'(i) = x(i) + g \wedge g' = g \\ \wedge \forall j \neq i . \big(\mathsf{loc}(j) = \mathsf{loc}'(j) \wedge x(j) = x'(j) \big) \Big).$$

We may now formally define Ashcroft invariants:

Definition 5.1 (Ashcroft invariant) Given a program P, an *Ashcroft invariant* is a global assertion Φ_{inv} such that

1. $\forall i . \mathsf{loc}(i) = \ell_{\mathsf{init}}$ entails Φ_{inv}.
2. For any command $\sigma \in \Sigma$, we have that $\Phi_{\mathsf{inv}} \wedge [\![\sigma]\!]$ entails Φ'_{inv} (where Φ'_{inv} denotes the formula obtained by replacing the symbols in Φ_{inv} with their primed copies)
3. Φ_{inv} entails $\forall i . \mathsf{loc}(i) \neq \ell_{\mathsf{err}}$. ⌟

Clearly, the existence of an Ashcroft invariant for a program implies its correctness (that is, that the error location of the program is unreachable). The following theorem is the main result of this section, which establishes the completeness of proof spaces relative to Ashcroft invariants.

Theorem 5.2 (Relative completeness) Let P be a program. If there is an Ashcroft invariant which proves the correctness of P, then there is a finitely generated proof space which proves the correctness of P. ⌟

Proof. To simplify notation, we will prove the result only for the case that the Ashcroft proof has two quantified thread variables. The generalization to an arbitrary number of quantified thread variables is straightforward.

Let P be a program, and let Φ_{inv} be an Ashcroft proof. Without loss of generality, we may assume that Φ_{inv} is written as

$$\Phi_{\mathsf{inv}} \equiv \big(\forall i \bigvee_{\ell \in \mathsf{Loc}} \mathsf{loc}(i) = \ell \wedge \varphi_\ell(i) \big) \\ \wedge \big(\forall i \neq j . \bigvee_{\ell, \ell' \in \mathsf{Loc} \times \mathsf{Loc}} \mathsf{loc}(i) = \ell \wedge \mathsf{loc}(j) = \ell' \wedge \varphi_{\ell,\ell'}(i, j) \big) \big)$$

where each $\varphi_\ell(i)$ and $\varphi_{\ell,\ell'}(i, j)$ is a integer arithmetic formula (i.e., does not contain loc). Furthermore, we can assume that Φ_{inv} is *symmetric* in the sense that $\varphi_{\ell,\ell'}(i, j)$ is syntactically equal to $\varphi_{\ell',\ell}(j, i)$ and that $\varphi_{\ell,\ell'}(i, j)$ entails $\varphi_\ell(i)$ (for any i, j, ℓ, ℓ'). We can prove that any Φ_{inv} can be written in this form by induction on Φ_{inv} We also assume that $\varphi_{\ell_{\mathsf{err}}}(i) = \mathit{false}$ and $\varphi_{\ell_{\mathsf{init}}}(i) = \varphi_{\ell_{\mathsf{init}},\ell_{\mathsf{init}}}(i, j) = \mathit{true}$; the fact that this loses no generality is a consequence of conditions 1 and 3 of Definition 5.1. Finally, we assume that each formula $\varphi_\ell(i)$ and $\varphi_{\ell,\ell'}(i, j)$ is not (syntactically) a conjunction (cf. Condition 1 of Definition 4.2): this assumption is validated by observing any formula φ is equivalent to $\varphi \vee \mathit{false}$, which is not a (syntactic) conjunction.

We construct a set of basic Hoare triples H by collecting the set of all Hoare triples of the following four types:

$$\{\varphi_{\ell_1}(1)\} \ \langle \sigma : 1 \rangle \ \{\varphi_{\ell'_1}(1)\}$$

$$\{\varphi_{\ell_1,\ell_2}(1, 2)\} \ \langle \sigma : 1 \rangle \ \{\varphi_{\ell_2}(2)\}$$

$$\{\varphi_{\ell_1,\ell_2}(1,2)\}\ \ \langle\sigma:1\rangle\ \ \{\varphi_{\ell_1',\ell_2}(1,2)\}$$

$\{\varphi_{\ell_1,\ell_2}(1,2)\wedge\varphi_{\ell_1,\ell_3}(1,3)\wedge\varphi_{\ell_2,\ell_3}(2,3)\}\ \langle\sigma:1\rangle\ \{\varphi_{\ell_2,\ell_3}(2,3)\}$ where $\mathsf{src}(\sigma)=\ell_1$ and $\mathsf{tgt}(\sigma)=\ell_1'$.

We now must show that (1) each Hoare triple in H is basic (i.e., satisfies Definition 4.2) and (2) for every trace τ of P, $\{\mathit{true}\}\ \ \tau\ \ \{\mathit{false}\}$ belongs to the proof space generated by H. Condition (2) is delayed to Section 6, where we prove a stronger result (Proposition 6.14). Here, we will just prove condition (1).

One can easily observe that conditions 1 and 2 of Definition 4.2 hold. It remains to show that each Hoare triple in H is valid. We will prove only the validity of the Hoare triple

$$\{\varphi_{\ell_1,\ell_2}(1,2)\}\ \ \langle\sigma:1\rangle\ \ \{\varphi_{\ell_1',\ell_2}(1,2)\}$$

where $\mathsf{src}(\sigma)=\ell_1$ and $\mathsf{tgt}(\sigma)=\ell_1'$. The other cases are similar.

Let us write $[\![\sigma]\!]$ as $\exists i.\psi^1(i)\wedge(\forall k.\psi^2(i,k))$. For a proof by contradiction, let us suppose that the above Hoare triple is invalid. This means:

$$\varphi_{\ell_1,\ell_2}(1,2)\wedge\psi^1(1)\wedge(\forall k.\psi^2(1,k))\not\models\varphi_{\ell_1',\ell_2}(1,2)'$$

It follows that there exists a structure $\mathfrak{A}$ such that

$$\mathfrak{A}\models\varphi_{\ell_1,\ell_2}(1,2)\wedge\psi^1(1)\wedge(\forall k.\psi^2(1,k))$$

and $\mathfrak{A}\not\models\varphi_{\ell_1',\ell_2}(1,2)'$. Without loss of generality, we may assume that $\mathsf{loc}(2)^{\mathfrak{A}}=\mathsf{loc}'(2)^{\mathfrak{A}}=\ell_2$. Our strategy will be to construct from $\mathfrak{A}$ a structure $\mathfrak{B}$ such that

$$\mathfrak{B}\models\Phi_{\mathsf{inv}}\wedge\psi^1(1)\wedge(\forall k.\psi^2(1,k))$$

but $\mathfrak{B}\not\models\Phi_{\mathsf{inv}}'$, which contradicts condition 2 of Definition 5.1.

We obtain $\mathfrak{B}$ simply by restricting the interpretation of the Thread sort to the set $\{1,2\}$ (intuitively: we consider a configuration of the program in which 1 and 2 are the only threads). Then we have

$$\mathfrak{B}\models\varphi_{\ell_1,\ell_2}(1,2)\wedge\psi^1(1)\wedge(\forall k.\psi^2(1,k))$$

by downward absoluteness of universal sentences and $\mathfrak{B}\not\models\varphi_{\ell_1',\ell_2}(1,2)'$ by upward absoluteness of quantifier-free sentences.

From $\mathfrak{B}\models\psi^1(1)$, we have $\mathfrak{B}\models\mathsf{loc}(1)=\ell_1$. By assumption, we have $\mathfrak{B}\models\mathsf{loc}(2)=\ell_2$. It follows that

$$\mathfrak{B}\models\varphi_{\ell_1,\ell_2}(1,2)\wedge\mathsf{loc}(1)=\ell_1\wedge\mathsf{loc}(2)=\ell_2$$

and thus, from the symmetry condition imposed on Φ_{inv}, that $\mathfrak{B}\models\Phi_{\mathsf{inv}}$. It is then easy to see that

$$\mathfrak{B}\models\Phi_{\mathsf{inv}}\wedge\psi^1(1)\wedge(\forall k.\psi^2(1,k))$$

holds. Finally, we note that since $\mathfrak{B}\models\mathsf{loc}'(1)=\ell_1'\wedge\mathsf{loc}'(2)=\ell_2$ and $\mathfrak{B}\not\models\varphi_{\ell_1',\ell_2}(1,2)$, $\mathfrak{B}$ is incompatible with every disjunct of

$$\bigvee_{\ell,\ell'\in\mathsf{Loc}\times\mathsf{Loc}}\mathsf{loc}'(1)=\ell\wedge\mathsf{loc}'(2)=\ell'\wedge\varphi'_{\ell,\ell'}(1,2)$$

and thus $\mathfrak{B}\not\models\Phi_{\mathsf{inv}}'$.

□

As we mentioned previously, Ashcroft invariants are able to make use of features which are not available to proof spaces, namely control assertions (i.e., assertions of the form $\mathsf{loc}(i)=\ell$) and universal quantification over thread variables. These "exotic" features are typical for program logics for concurrent programs with unbounded parallelism. As a general rule, classical verification techniques for sequential programs do not synthesize assertions which make use of these features. The price we pay for the relative ease of generating proof spaces is the relative difficulty of checking them. We address this topic in the next section.

6. Proof checking

We now turn to the main algorithmic problem suggested by the proof rule for proof spaces presented in Section 4: *given a program P and a finite basis H of a proof space $\mathscr{H}$, how do we check whether for every error trace τ we have that $\{\mathit{true}\}\ \ \tau\ \ \{\mathit{false}\}$ belongs to $\mathscr{H}$?* Our solution to this problem begins by introducing a new class of automata, *predicate automata*, which can be used to represent both the set of error traces of a program and the set of traces τ which have the infeasibility theorem $\{\mathit{true}\}\ \ \tau\ \ \{\mathit{false}\}$ in a finitely generated proof space. We show that the problem of checking whether for every error trace τ we have that $\{\mathit{true}\}\ \ \tau\ \ \{\mathit{false}\}$ belongs to $\mathscr{H}$ can be reduced to the emptiness problem for predicate automata. We show that the general emptiness checking problem is undecidable, and we give a semi-algorithm and show that it is a decision procedure for an interesting sub-class of predicate automata which correspond to thread-modular proofs.

6.1 Predicate automata

Predicate automata are a class of infinite-state automata which recognize languages over an infinite alphabet of the form $\Sigma\times\mathbb{N}$.[1] For readers familiar with alternating finite automata (AFA) [8, 9], a helpful analogy might be that predicate automata are to first-order logic what AFA are to propositional logic. A predicate automaton (PA) is equipped with a relational *vocabulary* $\langle Q,ar\rangle$ (in the usual sense of first-order logic) consisting of a finite set of predicate symbols Q and a function $ar:Q\to\mathbb{N}$ which maps each predicate symbol to its arity. A state of a PA is a proposition $q(\mathtt{i}_1,...,\mathtt{i}_{ar(q)})$, where $q\in Q$ and $\mathtt{i}_1,...,\mathtt{i}_{ar(q)}\in\mathbb{N}$. The transition function maps such states to formulas in the vocabulary of the PA (where disjunction corresponds to nondeterministic (existential) choice, and conjunction corresponds to universal choice). It is important to note that the symbols $q\in Q$ are "uninterpreted": they have no special semantics, and any subset of $\mathbb{N}^{ar(q)}$ is a valid interpretation of q.

Given a vocabulary $\langle Q,ar\rangle$ (and given the set TV of thread variables, i.e., variables whose values range over the set of thread identifiers $\mathbb{N}$), we define the set of *positive formulas* $\mathcal{F}(Q,ar)$ over $\langle Q,ar\rangle$ to be the set of negation-free formulas where each atom is either (1) a proposition of the form $q(i_1,...,i_n)$ (where $i_1,...,i_n\in\mathsf{TV}$), or (2) an equation $i=j$ (where $i,j\in\mathsf{TV}$), or (3) a disequation $i\neq j$ (where $i,j\in\mathsf{TV}$). Predicate automata are defined as follows:

Definition 6.1 (Predicate automata) A *predicate automaton* (PA) is a 6-tuple $A=\langle Q,ar,\Sigma,\delta,\varphi_{\mathsf{start}},F\rangle$ where

- $\langle Q,ar\rangle$ is a relational vocabulary
- Σ is a finite alphabet
- $\varphi_{\mathsf{start}}\in\mathcal{F}(Q,ar)$ is an initial formula with no free variables.
- $F\subseteq Q$ is a set of accepting predicate symbols
- $\delta:Q\times\Sigma\to\mathcal{F}(Q,ar)$ is a transition function which satisfies the property that for any $q\in Q$ and $\sigma\in\Sigma$, the free variables of $\delta(q,\sigma)$ are members of the set $\{i_0,...,i_{ar(q)}\}$. ⌟

To understand the restriction on the variables in the formula $\delta(q,\sigma)$, it may be intuitively helpful to think of q as $q(i_1,...,i_{ar(q)})$ and of σ as $\langle\sigma:i_0\rangle$.

We will elucidate this definition by first describing the dynamics of a PA. PA dynamics will be defined by a nondeterministic[2] transition system where edges are labeled by elements of the indexed

[1] Such languages are commonly called *data languages* [32].

[2] Readers familiar with AFA should note that we are effectively describing the "nondeterminization" of PA.

alphabet $\Sigma \times \mathbb{N}$, and where the nodes of the transition system are *configurations* which we will introduce next:

Definition 6.2 (Configuration) Let $A = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a PA. A *configuration* $\mathcal{C}$ of A is finite set of ground propositions of the form $q(\mathtt{i}_1, ..., \mathtt{i}_{ar(q)})$, where $q \in Q$ and $\mathtt{i}_1, ..., \mathtt{i}_{ar(q)} \in \mathbb{N}$. ⌟

It is convenient to identify a configuration $\mathcal{C}$ with the formula $\bigwedge_{q(\mathtt{i}_1,...,\mathtt{i}_{ar(q)})\in\mathcal{C}} q(\mathtt{i}_1, ..., \mathtt{i}_{ar(q)})$. We define the *initial* configurations of A to be the cubes of the disjunctive normal form (DNF) of φ_{start} (for example, if φ_{start} is $p \wedge (q \vee r)$, then the initial configurations are $p \wedge q$ and $p \wedge r$). A configuration is *accepting* if for all $q(\mathtt{i}_1, ..., \mathtt{i}_{ar(q)}) \in \mathcal{C}$, we have $q \in F$; otherwise, it is *rejecting*.

A PA $A = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F\rangle$ induces a transition relation on configurations as follows:

$$\mathcal{C} \xrightarrow{\sigma:\mathtt{k}} \mathcal{C}'$$

if $\mathcal{C}'$ is a cube in the DNF of the formula

$$\bigwedge_{q(\mathtt{i}_1,...,\mathtt{i}_{ar(q)})\in\mathcal{C}} \delta(q,\sigma)[i_0 \mapsto \mathtt{k}, i_1 \mapsto \mathtt{i}_1, ..., i_{ar(q)} \mapsto \mathtt{i}_{ar(q)}]$$

The fact that the free variables of $\delta(q, \sigma)$ must belong to the set $\{i_0, ..., i_{ar(q)}\}$ guarantees that the formula above has no free variables, and therefore its DNF corresponds to a set of configurations. Note also that the formula above may contain equalities and disequalities, but since they are ground (have no free variables), they are equivalent to either *true* or *false*, and thus can be eliminated.

We can think of $\delta(q, \sigma)$ as a rewrite rule whose application instantiates the (implicit) formal parameters $i_1, ..., i_{ar(q)}$ of q to the actual parameters $\mathtt{i}_1, ..., \mathtt{i}_n$ and instantiates i_0 to $\mathtt{k}$ (the index of the letter being read). In light of this interpretation, we will often write δ in a form that makes the (implicit) formal parameters explicit: for example, instead of

$$\begin{aligned}\delta(q, \sigma) =&(i_0 \neq i_1 \wedge (q(i_0, i_1) \vee q(i_1, i_2)))\\ &\vee (i_0 = i_1 \wedge q(i_1, i_2) \wedge q(i_2, i_1))\end{aligned}$$

we will typically write

$$\begin{aligned}\delta(q(i, j), \langle\sigma : k\rangle) =&(k \neq i \wedge (q(k, i) \vee q(i, j)))\\ &\vee (k = i \wedge q(i, j) \wedge q(j, i))\,.\end{aligned}$$

A trace $\tau = \langle\sigma_1 : \mathtt{i}_1\rangle \cdots \langle\sigma_n : \mathtt{i}_n\rangle$ is accepted by A if there is a sequence of configurations $\mathcal{C}_n, ..., \mathcal{C}_0$ such that:

1. $\mathcal{C}_n$ is initial
2. for each $r \in \{1, ..., n\}$, $\mathcal{C}_r \xrightarrow{\sigma_r:\mathtt{i}_r} \mathcal{C}_{r-1}$
3. $\mathcal{C}_0$ is accepting

It is important to note that the definition of acceptance implies that a PA reads its input from right to left rather than left to right. We will discuss the reason behind this when we explain the correspondence between proof spaces and predicate automata (Proposition 6.4).

Our first example of a predicate automaton will be the one constructed to accept the language of error traces of a program:

Proposition 6.3 Given a program P, there is a predicate automaton A_P such that $\mathcal{L}(A_P)$ is the set of error traces of P. ⌟

Proof. Let P be a program given by the thread template

$$P = \langle \mathsf{Loc}, \Sigma, \ell_{\mathsf{init}}, \ell_{\mathsf{err}}, \mathsf{src}, \mathsf{tgt}\rangle\,.$$

We define a PA $A_P = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F\rangle$, which closely mirrors the (reversed) control structure of P.

- $Q = \{\mathsf{loc}, \mathsf{err}\} \cup \mathsf{Loc}$, where err and loc are distinguished predicate symbols, to be explained in the following.
- $ar(\mathsf{loc}) = ar(\mathsf{err}) = 0$ and for all $\ell \in \mathsf{Loc}$, $ar(\ell) = 1$
- Let $\sigma \in \Sigma$ with, say, $\ell_1 = \mathsf{src}(\sigma)$ and $\ell_2 = \mathsf{tgt}(\sigma)$.

  The transition rule for a location $\ell \in \mathsf{Loc}$ is given by

$$\delta(\ell(i), \langle\sigma : j\rangle) = \begin{cases}(i = j \wedge \ell_1(i)) \vee (i \neq j \wedge \ell(i)) & \text{if } \ell = \ell_2\\ i \neq j \wedge \ell(i) & \text{if } \ell \neq \ell_2\end{cases}$$

  The transition rule for loc is given by

$$\delta(\mathsf{loc}, \langle\sigma : i\rangle) = \mathsf{loc} \wedge \ell_1(i)$$

  The transition rule for err is given by

$$\delta(\mathsf{err}, \langle\sigma : i\rangle) = \begin{cases}\ell_1(i) & \text{if } \ell_2 = \ell_{\mathsf{err}}\\ \mathsf{err} & \text{if } \ell_2 \neq \ell_{\mathsf{err}}\end{cases}$$

- $\varphi_{\mathsf{start}} = \mathsf{loc} \wedge \mathsf{err}$
- $F = \{\ell_{\mathsf{init}}, \mathsf{loc}\}$

Intuitively, the distinguished predicate symbol err represents "some thread is at the error location." The loc predicate is responsible for "initializing" the program counter of threads (in the backwards direction). That is, loc ensures that in every reachable configuration $\mathcal{C}$ of A_P, every $\sigma \in \Sigma$ and every $\mathtt{i} \in \mathbb{N}$, we have that if $\mathcal{C} \xrightarrow{\sigma:\mathtt{i}} \mathcal{C}'$, then $\ell_1(\mathtt{i}) \in \mathcal{C}'$, where $\ell_1 = \mathsf{src}(\sigma)$. For example, let $\sigma \in \Sigma$ with $\ell_1 = \mathsf{src}(\sigma)$ and $\ell_2 = \mathsf{tgt}(\sigma)$. By reading $\langle\sigma : 3\rangle$, the initial configuration $\mathsf{loc} \wedge \mathsf{err}$ may transition to $\ell_1(3) \wedge \mathsf{loc} \wedge \mathsf{err}$ if ℓ_2 is not ℓ_{err} or to $\ell_1(3) \wedge \mathsf{loc}$ if ℓ_2 is ℓ_{err}.

We omit a formal proof that $\mathcal{L}(A_P)$ is indeed the set of error traces of P. □

As the following proposition states, predicate automata are also sufficiently powerful to represent the set of traces which are proved correct by a given finitely-generated proof space.

Proposition 6.4 Let $\mathscr{H}$ be a proof space which is generated by a finite set of basic Hoare triples H. There exists a PA A_H (which can be computed effectively from H) such that $\mathcal{L}(A_H)$ is exactly the set of traces τ such that $\{\mathit{true}\}\ \tau\ \{\mathit{false}\} \in \mathscr{H}$. ⌟

Proof. Let H be a set of basic Hoare triples. The predicate automaton $A_H = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F\rangle$ closely mirrors the structure of H: intuitively, the predicates of A_H correspond to the assertions used in H, and each Hoare triple corresponds to a transition in δ.

The key step in defining A_H is to show how each Hoare triple in H corresponds to a transition rule (noting that if there are several Hoare triples with the same postcondition, we may combine their transition rules disjunctively). For a concrete example, consider the Hoare triple:

$$\{\mathtt{t(3)} \geq 0 \wedge \mathtt{t(9)} > \mathtt{t(3)}\}\ \langle \mathtt{t := 2*t} : 9\rangle\ \{\mathtt{t(9)} > \mathtt{t(3)}\}$$

This triple corresponds to the transition

$$\begin{aligned}\delta([\mathtt{t(1)} > \mathtt{t(2)}](i, j), \sigma : k) = i \neq j \wedge i = k \wedge j \neq k\\ \wedge [\mathtt{t(1)} \geq 0](j) \wedge [\mathtt{t(1)} > \mathtt{t(2)}](k, j)\end{aligned}$$

The predicates which appear in this formula are *canonical names* for the formulas in the Hoare triples (e.g., $[\mathtt{t(1)} \geq 0]$ is the canonical name for $\mathtt{t(3)} \geq 0$).

After constructing the transition relation as in the example, we construct A_H by taking the set of predicates to be the canonical names for formulas which appear in H, $[\mathit{false}]$ to be the initial formula, and $\{[\mathit{true}]\}$ to be the set of final formulas.

□

The construction of a PA from a set of basic Hoare triples for Proposition 6.4 reveals that the reason we defined predicate automata to read a trace backwards (i.e., the sequence of indexed

commands from right to left) is the asymmetry between pre- and postconditions in basic Hoare triples. Definition 4.2 requires that the postcondition of a basic Hoare triple cannot be constructed by conjoining two other formulas, while the precondition is arbitrary (i.e., we may think of the postcondition as a single proposition, while the precondition is a set of propositions). Since the transition function of a PA is defined on single propositions, the action of a transition must transform a postcondition to its precondition, which necessitates reading the traces backwards.

Propositions 6.3 and 6.4 together imply that the problem of checking whether a proof space proves the correctness of every trace of a program can be reduced to the language inclusion problem for predicate automata. The following proposition reduces the problem further to the emptiness problem for predicate automata (noting that $\mathcal{L}(A_P) \subseteq \mathcal{L}(A_H)$ is equivalent to $\mathcal{L}(A_P) \cap \overline{\mathcal{L}(A_H)} = \emptyset$):

Proposition 6.5 Predicate automata languages are closed under intersection and complement. ⌟

Proof. The constructions for intersection and complementation of predicate automata follow the classical ones for alternating finite automata.

Let A and A' be PAs. We form their intersection $A \cap A'$ by taking the vocabulary to be the disjoint union of the vocabularies of A and A', and define the transition relation and accepting predicates accordingly. The initial formula is obtained by conjoining the initial formulas of A and A'.

Given a PA $A = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F \rangle$, we form its complement $\overline{A} = \langle \overline{Q}, \overline{ar}, \Sigma, \overline{\delta}, \overline{\varphi_{\mathsf{start}}}, \overline{F} \rangle$ as follows.

We define the vocabulary $(\overline{Q}, \overline{ar})$ to be a "negated copy" of (Q, ar): $\overline{Q} = \{\overline{q} : q \in Q\}$ and $\overline{ar}(\overline{q}) = ar(q)$. The set of accepting predicate symbols is the (negated) set of rejecting predicate symbols from A: $\overline{F} = \{\overline{q} \in \overline{Q} : q \in Q \setminus F\}$. For any formula φ in $\mathcal{F}(Q, ar)$, we use $\overline{\varphi}$ to denote the "De Morganization" of φ, defined recursively by:

– $\overline{q(\vec{i})} = \overline{q}(\vec{i})$
– $\overline{i = j} = i \neq j$ and $\overline{i \neq j} = i = j$
– $\overline{\varphi \wedge \psi} = \overline{\varphi} \vee \overline{\psi}$ and $\overline{\varphi \vee \psi} = \overline{\varphi} \wedge \overline{\psi}$

We define the transition function and initial formula of $\overline{A}$ by De Morganization: $\overline{\delta}(\overline{q}, \sigma) = \overline{\delta(q, \sigma)}$ and the initial formula is $\overline{\varphi_{\mathsf{start}}}$. □

6.2 Checking emptiness for predicate automata

In this section, we give a semi-algorithm for checking PA emptiness which is sound (when the procedure terminates, it gives the correct answer) and complete for counter-examples (if the PA accepts a word, the procedure terminates). Our procedure for checking PA language emptiness is a variant of the tree-saturation algorithm for well-structured transition systems (cf. [18]). The algorithm is essentially a state-space exploration of a predicate automaton (starting from an initial configuration, searching for a reachable accepting configuration), but with one crucial improvement: we equip the state space with a *covering* pre-order, and prune the search space by removing all of those configurations which are not minimal with respect to this order. The key insight behind the development of well-structured transition systems is that, if the order satisfies certain conditions (namely, it is a well-quasi order[3]) this pruning strategy is sufficient to ensure termination of the search (i.e., although the search space may be infinite, the pruned search space is finite).

[3] A well-quasi order (wqo) is a preorder $\preceq$ such that for any infinite sequence $\{x_i\}_{i \in \mathbb{N}}$ there exists $i < j$ such that $x_i \preceq x_j$.

We begin by defining the covering relation on PA configurations:

Definition 6.6 (Covering) Given a PA $A = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F \rangle$, we define the *covering pre-order* $\preceq$ on the configurations of A as follows: if $\mathcal{C}$ and $\mathcal{C}'$ are configurations of A, then $\mathcal{C} \preceq \mathcal{C}'$ ("$\mathcal{C}$ *covers* $\mathcal{C}'$") if there is a permutation $\pi : \mathbb{N} \to \mathbb{N}$ such that for all $q \in Q$ and all $q(\mathtt{i}_1, .., \mathtt{i}_{ar(q)}) \in \mathcal{C}$, we have $q(\pi(\mathtt{i}_1), ..., \pi(\mathtt{i}_{ar(q)})) \in \mathcal{C}'$. ⌟

The idea behind the pruning strategy is that if two configurations $\mathcal{C}$ and $\mathcal{C}'$ are both in the search space and $\mathcal{C} \preceq \mathcal{C}'$, then we may remove $\mathcal{C}'$. The correctness of this strategy relies on the fact that if $\mathcal{C} \preceq \mathcal{C}'$ and an accepting configuration is reachable from $\mathcal{C}'$, then an accepting configuration is reachable from $\mathcal{C}$ (and thus, if an accepting configuration is reachable, then an accepting configuration is reachable without going through $\mathcal{C}'$). This fact follows from a downwards compatibility lemma:

Lemma 6.7 (Downwards compatibility) Let A be a PA and let $\mathcal{C}$ and $\mathcal{C}'$ be configurations of A such that $\mathcal{C} \preceq \mathcal{C}'$. Then we have the following:

1. If $\mathcal{C}'$ is accepting, then $\mathcal{C}$ is accepting.
2. For any $\langle \sigma : \mathtt{j} \rangle \in \Sigma \times \mathbb{N}$, if we have
$$\mathcal{C}' \xrightarrow{\sigma:\mathtt{j}} \overline{\mathcal{C}}'$$
then there exists a configuration $\overline{\mathcal{C}}$ and an index $\mathtt{k} \in \mathbb{N}$ such that
$$\mathcal{C} \xrightarrow{\sigma:\mathtt{k}} \overline{\mathcal{C}}$$
and $\overline{\mathcal{C}} \preceq \overline{\mathcal{C}}'$. ⌟

We now develop our algorithm in more detail. In the remainder of this section, let us fix a predicate automaton $A = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F \rangle$.

State-space exploration of PA is complicated by the fact that the alphabet $\Sigma \times \mathbb{N}$ is infinite and therefore PAs are infinitely branching (although for a fixed letter, each configuration has only finitely many successors). The key to solving this problem is that all but finitely many $\mathtt{i} \in \mathbb{N}$ are indistinguishable from the perspective of a given configuration $\mathcal{C}$. With this in mind, let us define the *support* $\mathsf{supp}(\mathcal{C})$ of a configuration $\mathcal{C}$ to be the set of all indices which appear in $\mathcal{C}$; formally,

$$\mathsf{supp}(\mathcal{C}) = \{\mathtt{i}_r : q(\mathtt{i}_1, ..., \mathtt{i}_{ar(q)}) \in \mathcal{C}, 1 \leq r \leq ar(q)\}$$

Intuitively, if $\mathtt{i}, \mathtt{j} \notin \mathsf{supp}(\mathcal{C})$, then $\mathtt{i}$ and $\mathtt{j}$ are effectively indistinguishable starting from $\mathcal{C}$. This intuition is formalized in the following lemma:

Lemma 6.8 Let $\mathcal{C}$ be a configuration, $\mathtt{k}_1, \mathtt{k}_2 \in \mathbb{N} \setminus \mathsf{supp}(\mathcal{C})$, and $\sigma \in \Sigma$. For all configurations $\mathcal{C}_1$ such that $\mathcal{C} \xrightarrow{\sigma:\mathtt{k}_1} \mathcal{C}_1$, there exists a configuration $\mathcal{C}_2$ such that $\mathcal{C} \xrightarrow{\sigma:\mathtt{k}_2} \mathcal{C}_2$ and $\mathcal{C}_1 \preceq \mathcal{C}_2$ and $\mathcal{C}_2 \preceq \mathcal{C}_1$. ⌟

As a result of this lemma, from a given configuration $\mathcal{C}$, it is sufficient to explore $\langle \sigma : \mathtt{i} \rangle$ such that $\mathtt{i} \in \mathsf{supp}(\mathcal{C})$, plus one additional $\mathtt{j} \notin \mathsf{supp}(\mathcal{C})$. In our algorithm we simply choose the additional $\mathtt{j}$ to be 1 more than the maximum index in $\mathsf{supp}(\mathcal{C})$.

Finally, we state our procedure for PA emptiness in Algorithm 1. This algorithm operates by expanding a reachability forest (N, E) where the nodes (N) are configurations and the edges (E) are labeled by indexed letters. The frontier of the reachability tree is kept in a worklist *worklist*, and the set of *closed* nodes (configurations which have already been expanded) is kept in $Closed$.

Theorem 6.9 Algorithm 1 is sound and is complete for non-emptiness: given a predicate automaton A, if Algorithm 1 returns Empty, then $\mathcal{L}(A)$ is $\emptyset$, and if $\mathcal{L}(A)$ is nonempty, then Algorithm 1 returns a word in $\mathcal{L}(A)$. ⌟

Input: predicate automaton $A = \langle Q, \Sigma, \delta, \varphi_{\mathsf{start}}, F \rangle$
Output: *Empty*, if $\mathcal{L}(A)$ is empty; a word $w \in \mathcal{L}(A)$, if not
Closed $\leftarrow \emptyset$;
$N \leftarrow \emptyset$;
$E \leftarrow \emptyset$;
worklist $\leftarrow \mathsf{dnf}(\varphi_{\mathsf{start}})$;
while *worklist* $\neq []$ **do**
  $\mathcal{C} \leftarrow$ *head*(*worklist*);
  worklist $\leftarrow$ *tail*(*worklist*);
  if $\neg\exists \mathcal{C}' \in$ *Closed s.t.* $\mathcal{C}' \preceq \mathcal{C}$ **then**
    `/* Expand` $\mathcal{C}$ `*/`
    foreach $\mathtt{i} \in \mathit{supp}(\mathcal{C}) \cup \{1 + \max \mathit{supp}(\mathcal{C})\}$ **do**
      foreach $\sigma \in \Sigma$ **do**
        foreach $\mathcal{C}'$ *s.t.* $\mathcal{C} \xrightarrow{\sigma:\mathtt{i}} \mathcal{C}'$ *and* $\mathcal{C}' \notin N$ **do**
          $N \leftarrow N \cup \{\mathcal{C}'\}$;
          $E \leftarrow E \cup \{\mathcal{C} \xrightarrow{\sigma:\mathtt{i}} \mathcal{C}'\}$;
          if $\mathcal{C}'$ *is accepting* **then**
            return *word* w *labeling the path in the graph* (N, E) *from* $\mathcal{C}'$ *to a root*;
          else
            worklist $\leftarrow$ *worklist* $+\!+ [\mathcal{C}']$;
          end
        end
      end
    end
  end
  Closed $\leftarrow$ *Closed* $\cup \{\mathcal{C}\}$;
end
return *Empty*

Algorithm 1: Emptiness check for predicate automata

6.3 Decidability results

Although Algorithm 1 is sound and complete for non-emptiness, it is *not* complete for emptiness: Algorithm 1 may fail to terminate in the case that the language of the input PA is empty. In fact, this must be the case for any algorithm, because PA emptiness is undecidable in the general case:

Proposition 6.10 General PA emptiness is undecidable. ⌟

Proof. The idea behind the proof is to reduce the halting problem for two-counter Minsky machines to the problem of deciding emptiness of a predicate automaton. The reduction uses two binary predicates, ln_1 and ln_2 to encode the value of the two counters: for example, a configuration

$$ln_1(4,3) \wedge ln_1(3,5) \wedge ln_1(5,1) \wedge ln_2(2,8)$$

encodes that the value of counter 1 is 3, while the value of counter 2 is 1 (i.e., the value of counter i corresponds to the length of the chain ln_i). The challenging part of this construction is to encode zero-tests, but we omit this technical discussion from the paper. □

Monadic predicate automata. Since PA emptiness is undecidable in general, it is interesting to consider subclasses where it is decidable. We say that a predicate automaton $A = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F \rangle$ is *monadic* if for all $q \in Q$, $ar(q) \leq 1$. We have the following:

Proposition 6.11 Algorithm 1 terminates for monadic predicate automata (i.e., emptiness is decidable for the class of monadic predicate automata, and Algorithm 1 is a decision procedure). ⌟

Proof. A sufficient (but not necessary) condition for Algorithm 1 to terminate is if $\preceq$ is a well-quasi order (wqo) - this is a standard result from well-structured transition systems. The fact that $\preceq$ is a well-quasi order on configurations of monadic predicate automata follows easily from Dickson's lemma.

The fact that Algorithm 1 is a decision procedure for the emptiness problem for monadic predicate automata follows from Theorem 6.9 and the fact that it terminates. □

The PA for a program (Proposition 6.3) always corresponds to a monadic PA; a finitely generated proof space $\mathscr{H}$ *fails* to be monadic (i.e., correspond to a monadic PA) exactly when one of the basic Hoare triples which generates it $\{\varphi\} \ \langle \sigma : \mathtt{i} \rangle \ \{\psi\}$ has a postcondition which relates the local variables of two or more threads together (for example, an assertion of the form $x(1) \leq y(2)$).

Monadic PA have a conceptual correspondence to thread-modular proofs [19], which also disallow assertions which relate the local variables of different threads. Indeed, if there exists a thread-modular proof for a program, then there exists a monadic proof space (i.e., a proof space which corresponds to a monadic PA). This correspondence is not exact, however: monadic proof spaces are strictly more powerful than thread-modular proofs. In particular, one can show that any correct Boolean (or finite-domain) program has a proof space which corresponds to a monadic proof space. To see why, consider that the transition function of a Boolean program corresponds to a finite set of Hoare triples.[4]

Discussion: decidability beyond monadic predicate automata. Note that the converse of Proposition 6.11 is not true, i.e., non-monadic proofs do not necessarily cause Algorithm 1 to diverge. For example, the proof of the thread pooling program from Section 2 is not monadic (for example, the assertion `end(1)` $\leq$ `c(2)` is dyadic). And yet, Algorithm 1 terminates for this example. We will informally discuss some other classes for which Algorithm 1 terminates.

We can generalize the monadic condition in a number of different ways while maintaining termination. One such generalization is *effectively monadic* PA, where there is a finite set of indices $D \subseteq \mathbb{N}$ such that in any reachable minimal (with respect to $\preceq$) configuration $\mathcal{C}$, for all $q(\mathtt{i}_1, ..., \mathtt{i}_n)$ we have at most one of $\mathtt{i}_1, ..., \mathtt{i}_n$ *not* in D. Intuitively, effectively monadic PAs can be used to reason about programs where one or more processes play a distinguished role (e.g., a client/server program, where there is a single distinguished server but arbitrarily many clients). Another alternative is *boundedly non-monadic* PA, where there exists some bound K such that in any reachable minimal (with respect to $\preceq$) configuration $\mathcal{C}$, the cardinality of the set $\{q(\mathtt{i}_1, ..., \mathtt{i}_n) \in \mathcal{C} : ar(q) > 1\}$ is less than K. The thread pooling example from Section 2 is boundedly non-monadic with a bound of 3. Intuitively, boundedly non-monadic PAs can be used to reason about programs which do not require "unbounded chains" of inter-thread relationships.

There is a great deal of research on proving that classes of systems are well-quasi ordered which can be adapted to the setting of predicate automata. For example, we may admit a single binary predicate which forms a total order relation (by Higman's lemma), or a tree (by Kruskal's tree theorem). Meyer showed in [30] that depth-bounded processes are well-quasi ordered, which implies that for depth-bounded processes of known depth, the covering problem can be decided using a standard backward algorithm for WSTSs. Intuitively, the covering problem asks whether a system can reach a configuration that contains some process that is

[4] This assumes that every command of the program is deterministic, but this is without loss of generality because for Boolean programs, it is always possible to replace nondeterministic commands (e.g., `b := *`) with a nondeterministic branch between deterministic commands (e.g., `b := 0` and `b := 1`).

in a local error state. The question whether the covering problem is decidable for the entire class of depth-bounded processes was later addressed in [39], where an adequate domain of limits was developed for well-structured transition systems that are induced by depth-bounded processes, and consequently the existence of a forward algorithm for deciding the covering problem was demonstrated.

6.4 Completeness of PA

In this section, we strengthen the relative completeness result from Section 5. Theorem 5.2 establishes that any Ashcroft proof for a program P corresponds to a finitely generate proof space $\mathscr{H}$ which covers the traces of P. But given our earlier undecidability result (Proposition 6.10), there is cause for concern that, although we can construct a basis H for $\mathscr{H}$, there may be no way to validate that it covers the traces of a program. In this section, we introduce emptiness certificates for PA, and complete the picture by showing that we can construct an emptiness certificate for $A_P \cap \overline{A_H}$.

Definition 6.12 Let $A = \langle Q, ar, \Sigma, \delta, \varphi_{\mathsf{start}}, F\rangle$ be a PA. An *emptiness certificate* for A is a formula $\varphi \in \mathcal{F}(Q, ar)$ (which may additionally have arbitrary quantification of thread variables) such that:

- $\varphi_{\mathsf{start}} \models \varphi$
- for all $\mathcal{C}, \mathcal{C}', \sigma, \mathtt{i}$ such that $\mathcal{C} \models \varphi$ and $\mathcal{C} \xrightarrow{\sigma:\mathtt{i}} \mathcal{C}'$, we have $\mathcal{C}' \models \varphi$.
- Every model of φ is rejecting. ⌟

An emptiness certificate can be seen as a kind of inductive invariant which shows that the language of a given predicate automaton is empty.

The following proposition establishes that emptiness certificates can be viewed as proofs that the language of a given predicate automaton is empty.

Proposition 6.13 Let A be a PA such that there exists an emptiness certificate for A. Then $\mathcal{L}(A) = \emptyset$. ⌟

Finally, we can state a strengthening of the completeness result from Section 5: not only do Ashcroft proofs correspond to finitely generated proof spaces, but they correspond to *checkable* proof spaces:

Proposition 6.14 (PA Completeness) Let P be a program. If there is an Ashcroft proof of correctness for P, then there is a proof space $\mathscr{H}$ which covers the traces of P, and there is an emptiness certificate for the predicate automaton $A_P \cap \overline{A_H}$. ⌟

Proof. We continue from the point started in the proof of Theorem 5.2: we let P be a program, Φ_{inv} be an Ashcroft invariant of the form

$$\Phi_{\mathsf{inv}} \equiv \big(\forall i.\ \bigvee_{\ell \in \mathsf{Loc}} \mathsf{loc}(i) = \ell \wedge \varphi_\ell(i)\big) \wedge \big(\forall i \neq j.\ \bigvee_{\ell,\ell' \in \mathsf{Loc}\times\mathsf{Loc}} \mathsf{loc}(i) = \ell \wedge \mathsf{loc}(j) = \ell' \wedge \varphi_{\ell,\ell'}(i,j)\big)\big)$$

and let H be as in Theorem 5.2.

We must construct an emptiness certificate for the automaton $A_P \cap \overline{A_H}$. Recall that the vocabulary of $A_P \cap \overline{A_H}$ consists of the vocabulary of A_P along with the "negated" vocabulary of A_H. That is, the set of predicate symbols is

$$\mathsf{Loc} \cup \{\mathsf{loc}, \mathsf{err}\} \cup \{\overline{\varphi_\ell} : \ell \in \mathsf{Loc}\} \cup \{\overline{\varphi}_{\ell,\ell'} : \ell, \ell' \in \mathsf{Loc}\} \cup \{\overline{[\mathit{false}]}\}$$

For any $\ell \in \mathsf{Loc}$ we use $\overline{\ell(i)}$ as shorthand for

$$\bigvee \{\ell'(i) : \ell' \in \mathsf{Loc} \wedge \ell \neq \ell'\}\ .$$

The following formula, which we call $\overline{\Phi}_{\mathsf{inv}}$, is such an emptiness certificate:

$$\mathsf{loc} \wedge \Big(\big(\exists i.\ \bigwedge_{\ell \in \mathsf{Loc}} \overline{\ell(i)} \vee \overline{\varphi_\ell}(i)\big) \vee \big(\exists i, j.i \neq j \wedge \bigwedge_{\ell,\ell' \in \mathsf{Loc}\times\mathsf{Loc}} \overline{\ell(i)} \vee \overline{\ell'(j)} \vee \overline{\varphi_{\ell,\ell'}}(i,j)\big)\big) \vee (\mathsf{err} \wedge \overline{[\mathit{false}]})\Big)$$

The conditions of Definition 6.12 can easily be checked. The intuition behind this emptiness certificate comes from the observation that the negation of a forwards inductive invariant is a backwards inductive invariant. □

7. Verification algorithm

In this section, we outline a verification algorithm based on the automatic construction of proof spaces, and then discuss how the algorithm can be customized using various heuristics known and some perhaps yet to be discovered. Keep in mind that our aim in this section is not the presentation and/or evaluation of such heuristics, but rather to place proof spaces in their algorithmic context and to discuss some of the interesting research problems which are posed by this specific algorithmic context.

The high-level verification procedure based on proof spaces is given in Algorithm 2. It is essentially a variation of a standard counter-example guided abstraction refinement (CEGAR) loop. $\mathsf{proof\text{-}space}(\tau)$ is a procedure that computes a finite set of basic Hoare triples such that $\{\mathit{true}\}\ \ \tau\ \ \{\mathit{false}\}$ belongs to the proof space generated by $\mathsf{proof\text{-}space}(\tau)$.

One straightforward implementation of proof-space is to use sequence interpolation [22]. Given a trace $\tau = \langle\sigma_1 : \mathtt{i}_1\rangle \cdots \langle\sigma_n : \mathtt{i}_n\rangle$, we may use sequence interpolation to compute a sequence of intermediate assertions $\varphi_1, ..., \varphi_{n+1}$ such that $\varphi_1 = \mathit{true}$, $\varphi_{n+1} = \mathit{false}$, and for every $j \leq n$, the Hoare triple $\{\varphi_j\}\ \ \langle\sigma_j : \mathtt{i}_j\rangle\ \ \{\varphi_{j+1}\}$ is valid. We may then take the $\mathsf{proof\text{-}space}(\tau)$ to be the set of all such Hoare triple $\{\varphi_j\}\ \ \langle\sigma_j : \mathtt{i}_j\rangle\ \ \{\varphi_{j+1}\}$.[5]

Algorithm 2 takes as input a program P and, if it terminates, returns either a counter-example τ showing that the error location of P is reachable or a basis H of a proof space which proves the correctness of P. The algorithm operates by repeatedly sampling error traces τ of P for which $\{\mathit{true}\}\ \ \tau\ \ \{\mathit{false}\}$ is not in the proof space generated by H. If τ is feasible then the program is incorrect and the counter-example τ is returned. Otherwise, we add the Hoare triples from $\mathsf{proof\text{-}space}(\tau)$ to H. If we are *unable* to sample a trace τ for which $\{\mathit{true}\}\ \ \tau\ \ \{\mathit{false}\}$ is not in the proof space generated by H, then H is a basis for a proof space which proves the correctness of P, and we return H. The high-level properties of this algorithm are summarized in the following theorem.

Theorem 7.1 Algorithm 2 is sound and is complete for counter-examples: given a program P, if Algorithm 2 returns Safe, then ℓ_{err} is unreachable; if ℓ_{err} is reachable, then Algorithm 2 returns a feasible trace which reaches ℓ_{err}. ⌟

Discussion There is a great deal of flexibility in the design choices of the proof-space procedure. Let us discuss some of the design considerations and directions for future research concerning the construction of a proof space from a trace. An abstract proof-space procedure can be viewed to include these two steps:

[5] Technically speaking, these triples may not satisfy condition 1 of Definition 4.2, so we must define $\mathsf{proof\text{-}space}(\tau)$ to be the set of all $\{\varphi_j \vee \mathit{false}\}\ \ \langle\sigma_j : \mathtt{i}_j\rangle\ \ \{\varphi_{j+1} \vee \mathit{false}\}$.

Construct program automaton A_P;
$H \leftarrow \emptyset$; /* Basis for a proof space */
while $(A_P \cap \overline{A_H}) \neq \emptyset$ **do** /* Algorithm 1 */
  Select τ from $A_P \cap \overline{A_H}$; /* Algorithm 1 */
  if τ *is feasible* **then**
    return *Counter-example* τ
  else
    $H \leftarrow H \cup \mathsf{proof\text{-}space}(\tau)$
  end
end
return *Proof* H

Algorithm 2: Verify(P)

1. Construct a program P' such that P' over-approximates the input trace τ (in the sense that τ is an error trace of P' and $\{true\}\ P'\ \{false\}$), and construct a proof of correctness for P'.
2. Decompose the proof from step 1 to get a finite set of basic Hoare triples.

Step 1 can be replaced by a variety of different algorithms. The straightforward algorithm we suggested above (i.e. sequence interpolants) is one option. One can also view Step 1 to be implemented as a refinement loop, in the sense that an approximation is constructed and if a proof of correctness cannot be constructed for it, it is refined until a proof can be found. There are an array of known techniques that lie in the middle of the spectrum from the sequence interpolants for the trace (where we take $P' = \tau$), to a fully generalized refinement construction of the generalization of the trace (where P' is the most general provably correct program containing τ). For example, path programs [6] can be used to restore some of the looping structure from P in order to take advantage of (sequential) loop invariant generation techniques. Bounded programs [38] are another category of suitable candidates for P', where a loop-free concurrent underapproximation of P is constructed through "de-interleaving" τ; a correctness proof for the bounded program can then be constructed using techniques for fixed-thread concurrent program verification.

Step 2, decomposition of the proof, is more subtle than it might seem. In particular, the requirement that preconditions of basic Hoare triples may contain only relevant threads (condition 2 of Definition 4.2) may be difficult to enforce. An interesting feature of the sequence interpolation procedure outlined above is that the relevance requirement follows immediately from the properties of interpolants. But in general, it may be necessary to develop non-trivial algorithms for decomposing a proof into basic Hoare triples.

An interesting alternative to step 2 is to consider the problem of constructing basic Hoare triples directly rather than to extract them from an existing proof. For example, one possibility is to design an interpolation procedure which yields monadic proof spaces, perhaps borrowing techniques from tree interpolation [10].

8. Related work

There is a huge body of work on analysis and verification of concurrent programs. In this section, we limit ourselves to the substantial body of work on verification of programs with unbounded parallelism. Below, we provide some context for our work in this paper, and compare with the most related work.

Unbounded parallelism and unbounded memory The proof checking procedure presented in Section 6 can be viewed as a kind of pre^* analysis using the proof space, which may be reminiscent of predicate abstraction. There are two particular approaches to predicate abstraction for programs with unbounded parallelism and unbounded memory which are related to ours: *indexed predicate abstraction* [28] and *dual reference programs* [25]. Indexed predicate abstraction allow predicates to have free (thread) variables (e.g., $x(i) \geq 0$), with the ultimate goal of computing an Ashcroft invariant (i.e., a universally quantified invariant) of a fixed quantifier depth. Admitting free thread variables allows indexed predicates to be combined "under the quantifier" and thus compute complex quantified invariants from simple components. In view of the symmetry closure condition of proof spaces, indexed predicates serve a similar function to our ground Hoare triples, and our use of combinatorial generalization shares the goal of [28] to compute complex invariants from simple components. [28] uses a theorem prover to reason about universal quantifiers and program data simultaneously, and determines an Ashcroft invariant for a program by computing the least fixpoint in a finite abstract domain determined by a set of indexed predicates. Our technique does not require a theorem prover which supports universal quantifiers (or heuristics for quantifier instantiation), and the predicate automata inclusion check algorithm (based on the tree-saturation algorithm for well-structured transition systems) replaces the abstract fixpoint computation. The separation between reasoning about data and thread quantification provides a fresh perspective on what exactly makes reasoning about unbounded parallelism difficult, and enables us to state and prove results such as the decidability of the monadic case (i.e. Proposition 6.11), which has no obvious analogue in the setting of [28].

Dual reference programs allow two types of predicates: *single-thread predicates* (*mixed assertions* in the terminology in [12]), which refer to globals and the locals of one thread, and *inter-thread predicates*, which refer to globals and the locals of two threads, where one thread is universally quantified (e.g., $\forall j.x(i) < x(j)$). Although dual reference programs are not necessarily well-structured transition systems, [25] shows that it is always possible to convert a dual reference program obtained from an asynchronous program via predicate abstraction to a WSTS. In contrast with [25], our technique is complete relative to Ashcroft invariants. Moreover, we are able to use standard techniques from sequential verification to compute refinement predicates, whereas it is less clear how to automatically generate their inter-thread predicates that are of the form $\varphi(l, l_P)$, where l is the local variable of a distinct active thread, and l_p stand for a local variable of *all* passive threads (and therefore the predicates are universally quantified).

Model checking modulo theories (MCMT) is a general method for proving safety of array-based systems [20], which generalize finite-state parameterized systems. The MCMT algorithm is essentially a pre^* computation which utilizes techniques from well-structured transition systems to guarantee termination for a subclass of array-based systems (which can be expressed in well-quasi ordered theories); this is similar in spirit to the proof checking algorithm presented in Section 6 and the covering relation defined in [20] is strikingly similar to Definition 6.6. In [3], MCMT is combined with interpolation-based abstraction, which in some practical cases terminates when the MCMT algorithm does not. Perhaps the most notable difference between our approach and MCMT is that we separate the verification problem into two sub-problems: constructing a proof space, and checking the adequacy of the proof space.

There is some work on automated invariant generation for programs where the number of threads and memory are unbounded [13, 23, 36]. These methods generally differ from ours in that they compute inductive invariants and are not property driven. In [36], starting with a set of candidate invariants (assertions), the approach builds a reflective abstraction, from which invariants of the concrete system are obtained in a fixpoint process. The number of quantifiers used in the invariants is a parameter of the abstract domain that needs to be fixed a priori. In [13], data flow graphs are used

to compute invariants of concurrent programs with unboundedly many threads. Their abstraction is not expressive enough to capture relations between local variables and global variables, or local variables of other threads. In [23], a language of recursively defined formulas about arrays of variables, suitable for specifying safety properties of parameterized systems, is used. Their main contribution is a proof method for implications between such formulas, which accommodates the implementation of an abstract interpreter. Their effort was in the direction of proving entailment between complex formulas; our effort is to avoid them altogether.

The problem of proving data structure invariants for programs with unboundedly many threads is attacked in [5] and [37]. [5] aims to exploit thread-modularity in their proofs, which restricts the ways in which thread-local variables may be correlated (for the practical gain of a faster analysis). Additional correlations can be captured using the technique of [37], in which a universally quantified *environment assertion* is used to keep track of relationships between a distinguished thread and all other threads.

Unbounded Parallelism and Bounded Memory In [1], a border case between unbounded and bounded data is investigated where shared variables range over unbounded domain of naturals but the local states of processes are finite (i.e. local variables of threads range over finite domains). There has been a great deal of work in the area of automated verification and analysis of concurrent programs where the number of threads is unbounded but the threads are finite-state [7, 24, 27, 31, 33]. We skip a detailed discussion of the wealth of techniques in this area, as they are not as closely related to our technique in the sense that they are limited to finite-state threads.

Automata on infinite alphabets One of the main contributions of this paper is a method for determining whether the language of error traces of some program P are included inside the set language of correct traces proved by some proof space $\mathscr{H}$. Classical automata-theoretic techniques cannot directly be applied because the alphabet of program commands is infinite. However, the automata community has developed generalizations of automata to infinite alphabets; the most relevant to our work is *(alternating) register automata (ARA)*, which are closely related to predicate automata.

Register automata were first introduced in [26]. Universality for register automata was shown to be undecidable in [32], which implies ARA emptiness is undecidable in the general case. However, the emptiness problem for ARA with 1 register (cf. monadic predicate automata) was proved to be decidable in [11] by reduction to reachability for lossy counter machines; and a direct proof based on well-structured transition systems was later presented in [16].

Language-theoretic program verification The method presented in this paper is inspired by the language-theoretic approach to program correctness proposed in [21]. Notably, this approach has also been used in the context of concurrent programs with a fixed number of threads [14] and concurrent programs with unboundedly many threads [15]. One fundamental difference between our approach and previous language-theoretic techniques is that in [14, 15, 21], a *finite* set of program statements and therefore a *finite* alphabet was used. In [15], counting proofs were presented as a method for automatically synthesizing auxiliary variables in complex counting arguments for parameterized protocols. Counting proofs, i.e. proofs that use auxiliary counters are not expressible as proof spaces without the use of these auxiliary variables. On the other hand, proof spaces are capable of proving properties of programs which involve reasoning about infinite-domain local variables which is beyond the counting proofs framework. It will be interesting to investigate whether the strengths of the counting proofs and proof spaces can be combined into a single framework.

9. Conclusion

We conclude with a discussion of our work in a wider context and with an outlook to future work.

Abstract interpretation From the perspective of abstract interpretation, proof spaces introduce a new class of abstract transformers for multi-threaded programs. We will here only sketch the details of a formalization of proof spaces in the abstract interpretation framework. The main part of the PA emptiness algorithm (Algorithm 1) applied to the predicate automaton $A_P \cap \overline{A_H}$ (for the program P and the basis H of a proof space) iterates an abstraction of the pre-image of the transition function. The corresponding abstract domain is the free lattice generated by the set consisting of the negations of the assertions that are used in the basis H and the set of control assertions. The fact that the lattice is free (i.e., ignores the logical meaning of assertions) justifies our calling the abstract fixpoint computation *combinatorial*. The definition of the abstract transformer can be inferred from the definition of the transition function of the predicate automaton.

In contrast to work on software model checking for unbounded parallelism, we have not phrased abstraction as a source-to-source transformation, i.e., a mapping of a program into another one. Instead, we have taken the more general approach of abstract interpretation, which is to define an abstraction through a mapping of a semantics-defining function over a partially-ordered domain into another one.

Symbolic vs. combinatorial reasoning In program verification, there is a tradition of dividing the problem of proving that a program satisfies a property of interest into two sub-tasks: (1) finding a simplified model of the program which simulates it, and (2) verifying that the property holds in the model. The first is a *symbolic* problem which requires reasoning about the program's (often infinite-domain) data theory (e.g., integers), and the second is *combinatorial* problem over a (possibly very large) finite state space. A classical example of this is the SLAM tool, which computes a recursive Boolean program which abstracts a given program using a finite set of predicates (generated via symbolic reasoning) and then verifies the Boolean program using CFL reachability [34] (a combinatorial algorithm).

Proof spaces achieve a similar separation between symbolic and combinatorial reasoning: symbolic reasoning is required to construct the basis of a proof space, and then checking whether every error trace can be proved infeasible using the sequencing, symmetry, and conjunction rules is a combinatorial problem. From this point of view, our algorithm for predicate automaton emptiness serves a role which is analogous to CFL reachability in SLAM. It is interesting to note that (unlike CFL reachability), the emptiness problem for general predicate automata is undecidable. There is a body of work on other combinatorial verification problems to serve as targets for multi-threaded programs [1, 25], but none are complete (in the sense of Section 5). Thus, our work reflects a fundamental tension between completeness and decidability in the setting of unbounded parallelism which does not exist for sequential programs or even multi-threaded programs with a fixed number of threads.

Unbounded vs. Infinite The number of threads which participate in an error trace of a multi-threaded program is *unbounded* rather than *infinite*. This enables proof spaces to use conjunction and symmetry in place of universal thread quantification, and thus avoid the problem of directly synthesizing quantifiers. One may view using universal quantification to reason about unbounded threads (as in Ashcroft's proof system) as a deficiency, because an entailment is only provable in first order logic if it holds for *all* models, including models which have an infinite universe of threads.

For termination and liveness properties, one needs to reason about infinite traces. This creates the need to distinguish between programs with unboundedly many threads (e.g., computations which are parallelized over a number of processes which is determined at run time) and programs with infinitely many threads (e.g., programs with dynamic thread creation which spawn infinitely many threads over an infinite execution). In the setting of unbounded parallelism, then even if a trace is infinite, it can involve only finitely many threads. Thus, it may in principle be possible to extend proof spaces to termination and liveness properties. This is a topic of future work.

References

[1] Parosh A. Abdulla, Yu-Feng Chen, Giorgio Delzanno, Frédéric Haziza, Chih-Duo Hong, and Ahmed Rezine. Constrained monotonic abstraction: a CEGAR for parameterized verification. In *CONCUR*, pages 86–101, 2010.

[2] Parosh A. Abdulla, Karlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321, 1996.

[3] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. Lazy abstraction with interpolants for arrays. In *LPAR*, pages 46–61, 2012.

[4] Edward A. Ashcroft. Proving assertions about parallel programs. *J. Comput. Syst. Sci.*, 10(1):110–135, February 1975.

[5] Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, pages 399–413, 2008.

[6] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI*, pages 300–309, 2007.

[7] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *CAV*, pages 403–418, 2000.

[8] Janusz A. Brzozowski and Ernst L. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10(1):19–35, 1980.

[9] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, January 1981.

[10] Jürgen Christ and Jochen Hoenicke. Extending proof tree preserving interpolation to sequences and trees. In *Workshop on SMT Solving*, 2013.

[11] Stéphane Demri and Ranko Lazić. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Logic*, 10(3):16:1–16:30, April 2009.

[12] 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):25–44, 2012.

[13] Azadeh Farzan and Zachary Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *POPL*, pages 297–308, 2012.

[14] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Inductive data flow graphs. In *POPL*, pages 129–142, 2013.

[15] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proofs that count. In *POPL*, pages 151–164, 2014.

[16] Diego Figueira. Alternating register automata on finite words and trees. *Logical Methods in Computer Science*, 8(1), 2012.

[17] Alain Finkel. A generalization of the procedure of Karp and Miller to well structured transition systems. In *ICALP*, pages 499–508, 1987.

[18] Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1):63–92, 2001.

[19] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Thread-modular verification for shared-memory programs. In *ESOP*, pages 262–277, 2002.

[20] Silvio. Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Towards SMT model checking of array-based systems. In *IJCAR*, pages 67–82, 2008.

[21] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In *SAS*, pages 69–85, 2009.

[22] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244, 2004.

[23] Joxan Jaffar and Andrew E. Santosa. Recursive abstractions for parameterized systems. In *FM*, pages 72–88, 2009.

[24] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, pages 645–659, 2010.

[25] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Lost in abstraction: Monotonicity in multi-threaded programs. In *CONCUR*, pages 141–155, 2014.

[26] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, November 1994.

[27] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV*, pages 629–644, 2010.

[28] Shuvendu K. Lahiri and Randal E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Logic*, 9(1), December 2007.

[29] Alexander Malkis. *Cartesian abstraction and verification of multi-threaded programs*. PhD thesis, University of Freiburg, 2010.

[30] Roland Meyer. On boundedness in depth in the pi-calculus. In *IFIP TCS*, pages 477–489, 2008.

[31] Kedar S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *VMCAI*, pages 299–313, 2007.

[32] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, July 2004.

[33] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, pages 82–97, 2001.

[34] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[35] Willem-Paul de Roever. *Concurrency verification: introduction to compositional and noncompositional methods*. Cambridge University press, Cambridge, 2001.

[36] Alejandro Sanchez, Sriram Sankaranarayanan, César Sánchez, and Bor-Yuh Evan Chang. Invariant generation for parametrized systems using self-reflection. In *SAS*, pages 146–163. Springer, 2012.

[37] Michal Segalov, Tal Lev-Ami, Roman Manevich, Ramalingam Ganesan, and Mooly Sagiv. Abstract transformers for thread correlation analysis. In *APLAS*, pages 30–46, 2009.

[38] Nishant Sinha and Chao Wang. On interference abstractions. In *POPL*, pages 423–434, 2011.

[39] Thomas Wies, Damien Zufferey, and Thomas A. Henzinger. Forward analysis of depth-bounded processes. In *FOSSACS*, pages 94–108, 2010.