Proofs That Count

Azadeh Farzan Zachary Kincaid
University of Toronto

Andreas Podelski
University of Freiburg

Abstract

Counting arguments are among the most basic proof methods in mathematics. Within the field of formal verification, they are useful for reasoning about programs with *infinite control*, such as programs with an unbounded number of threads, or (concurrent) programs with recursive procedures. While counting arguments are common in informal, hand-written proofs of such programs, there are no fully *automated* techniques to construct counting arguments. The key questions involved in automating counting arguments are: *how to decide what should be counted?*, and *how to decide when a counting argument is valid?* In this paper, we present a technique for automatically constructing and checking counting arguments, which includes novel solutions to these questions.

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

General Terms Languages, Verification

Keywords Concurrency, Verification, Static Analysis

1. Introduction

A *counting argument* (in the context of formal methods) is a program proof that makes use of one or more *counters*, which are not part of the program itself, but which are useful for abstracting program behaviour. For example it may be useful to refer to the number of threads that have requested access to a shared resource, or the number of times a recursive procedure has been called. Despite the usefulness of counting arguments in hand-written proofs, the problem of constructing such arguments automatically is little explored. This paper presents one such approach.

The main intellectual challenge of constructing counting arguments automatically is that we must design an algorithm which can choose what to count. This is a task that, when carried out by humans, seems to require genuine creativity. The question of whether machines are capable of simulating this type of creativity is a challenging problem in formal verification. It is also a *fundamental* one: counters may be viewed as a class of *auxiliary variables* (*á la* Owicki-Gries [41]), in the sense that they remember useful information about the program history that can be used in a formal argument. The problem of "choosing what to count" can be viewed as one incarnation of the well-known (and yet under-explored) problem of auxiliary variable synthesis.

This paper presents a strategy for automatically constructing counting arguments for program verification. This strategy is outlined in Figure 1. The input to our algorithm is a program $\mathcal{P}$ and a pre/postcondition specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$. The algorithm can be viewed as a kind of language inference algorithm. The program $\mathcal{P}$ is treated as a black box: we can sample traces from $\mathcal{P}$, but we may not inspect its internal structure (so effectively, we identify $\mathcal{P}$ with a language of traces). The goal of our algorithm is to learn a *counting proof* $\langle A, \varphi \rangle$ for $\mathcal{P}$. A counting proof consists of a *counting automaton* A, which determines a language of traces, and an annotation φ, which is a proof that all traces in the language of A satisfy the specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$. If our algorithm can learn a counting proof $\langle A, \varphi \rangle$ such that A recognizes all the traces from $\mathcal{P}$, then $\mathcal{P}$ is correct with respect to the specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$.

The algorithm in Figure 1 operates as follows. We start by (a) constructing a counting proof $\langle A, \varphi \rangle$ for set of sample traces Tr. We then (b) check if every program trace in $\mathcal{P}$ is recognized by the counting automaton A. If the check succeeds, the algorithm has learned a counting proof for $\mathcal{P}$. If it fails, it produces a counterexample τ (a program trace that is not recognized by A). We then (c) check whether τ satisfies the specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$. If not, we are done: τ is a counterexample which shows that $\mathcal{P}$ is incorrect. If yes, then (d) we add τ to the set of sample traces Tr, and repeat. Note that in the first iteration (since initially Tr is empty) the language of A is empty, so τ is any program trace.

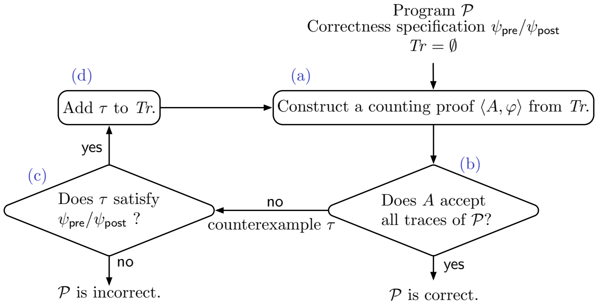
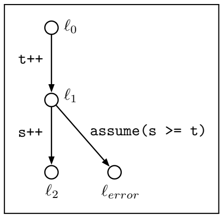

Program $\mathcal{P}$
Correctness specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$
$Tr = \emptyset$
(d)
Add τ to Tr.
(a)
Construct a counting proof $\langle A, \varphi \rangle$ from Tr.
yes
(c)
Does τ satisfy $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$?
no
counterexample τ
(b)
Does A accept all traces of $\mathcal{P}$?
no
$\mathcal{P}$ is incorrect.
yes
$\mathcal{P}$ is correct.

Figure 1. Counting proof inference.

Before we explain counting proofs further, we remark on the important distinction between *discovery* and *synthesis* of auxiliary variables (and counters in particular). Counter discovery is essentially a white-box technique: the goal is to expose existing counters in a program which are relevant for a proof (for example, discovery of relevant program counters for Owicki-Gries proofs, as in [23]). Counter synthesis is a black-box technique: the goal is to discover new counters which are useful for a proof, which do not necessarily correspond to anything in the program. This paper addresses the latter problem. This is the motivation behind our

POPL'14, January 22 - 24 2014, San Diego, CA, USA.
Copyright © 2014 ACM 978-1-4503-2544-8/14/01...$15.00.
http://dx.doi.org/10.1145/2535838.2535885

language-theoretic model of program correctness: by treating the $\mathcal{P}$ as a language of traces, we are forced to synthesize counting arguments from scratch. Our interest in counter synthesis stems from the fact that the internal control structure of $\mathcal{P}$ may be very complicated. Counter synthesis gives a strategy for constructing proofs for $\mathcal{P}$ that avoids reasoning about its control and data simultaneously.



We will explain counting proofs and our proof inference algorithm by way of an example. Consider the program that consists of an arbitrary number of threads whose control flow graph is pictured to the right. The (global) integer variables s and t are initially 0. The task is to automatically construct a proof that the error location ℓ_{error} is unreachable (i.e., the program satisfies the specification $s = t = 0/false$). The task is complicated by the fact that the seemingly simple program has a complex control structure, since it must retain the control location for every thread (of which there are unboundedly many).

We begin by sampling an error trace from this program, say

$$\tau = \texttt{t++; t++; s++; assume(s >= t)}$$

A correctness proof for τ is a sequence of intermediate assertions, shown below in Figure 2(a). This proof can be generalized to a proof for a whole language of program traces. This is done by constructing a finite automaton annotated with a Floyd/Hoare proof (with one state for each distinct assertion) as pictured in Figure 2(b). It is easy to see that any trace accepted by this automaton satisfies the given specification.

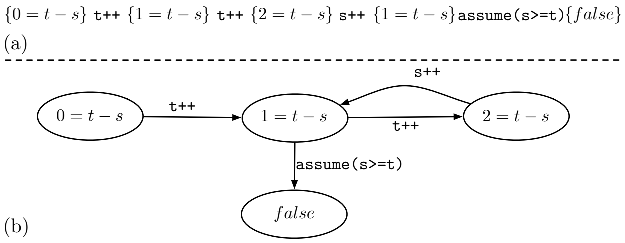
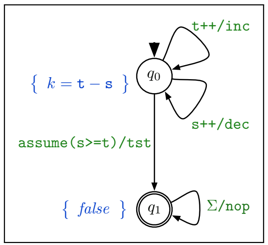

$\{0 = t - s\}$ t++ $\{1 = t - s\}$ t++ $\{2 = t - s\}$ s++ $\{1 = t - s\}$assume(s>=t)$\{false\}$
(a)
$0 = t - s$
t++
$1 = t - s$
s++
t++
$2 = t - s$
assume(s>=t)
$false$
(b)

Figure 2. Proof for the sample trace τ.

Unfortunately, this automaton does not accept every trace of the program. We could continue by sampling a new trace, for instance

$$\tau' = \texttt{t++; t++; t++; s++; assume(s >= t)},$$

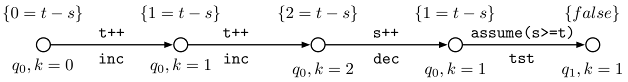
but it is already clear that this strategy is doomed to fail. There is no regular language which contains all the program traces and which does not contain incorrect traces.



Our solution to this problem is pictured to the right. This *counting proof* consists of a counting automaton A (a kind of restricted counter machine) paired with an annotation φ mapping the states of A to assertions. The counting automaton A is a finite automaton equipped with a $\mathbb{N}$-valued counter denoted k (initially 0). Each transition of the automaton is equipped with an action for k, which may be `inc` (increment the counter), `dec` (decrement, but block unless the counter is ≥ 1), `tst` (block unless the counter is ≥ 1), or `nop` (do nothing). The annotation φ associates with each state of this automaton a formula over the program variables and the counter variable k. This annotation is *inductive* in the sense that each transition is associated with a valid Hoare triple: for example,

$\{k = t - s\}$	`t++; k++`	$\{k = t - s\}$
$\{k = t - s\}$	`s++; k--`	$\{k = t - s\}$
$\{k = t - s\}$	`assume(s>=t); assume(k`$\geq$`1)`	$\{false\}$

A trace is accepted by A if it labels a path from q_0 (the initial state) to q_1 (the final state), and none of the counter actions block. Every trace which is accepted by A is associated with a sequence of intermediate assertions which prove its correctness. This sequence is obtained from the accepting run of A by taking, for each position in the run, the assertion at the current state with k replaced by its current value. For example, the proof for the trace τ above is as follows:



This counting proof works not only for the trace τ, but for *every* trace of the program (that is, the proof is enough to show that ℓ_{error} is unreachable). The key to this proof is the use of the counter variable k, which intuitively counts the number of `t++` statements in excess of `s++` statements along a trace. Using this auxiliary counter allows us to make a simple, succinct argument for the correctness of this program.

There are two algorithmic problems associated with our proof inference strategy: (1) how to construct a counting proof for a set of sample traces (Figure 1 (a)), and (2) how to prove that a counting proof recognizes all program traces (Figure 1 (b)).

The essential idea for our solution to (1) is to encode the proof construction problem as an SMT query. Our encoding requires us to specify the "size" of the candidate proof to find (e.g., the number of states that may be used), and will always succeed if a proof of that size exists. The main insight behind our proof construction procedure is that by looking for *small* proofs, we can force an SMT solver to synthesize nontrivial counting arguments. For example, we can force an SMT solver to "discover" the need to count the number of `t++` statements in excess of `s++` statements in the proof above completely automatically, simply by asking for a proof with 2 states.

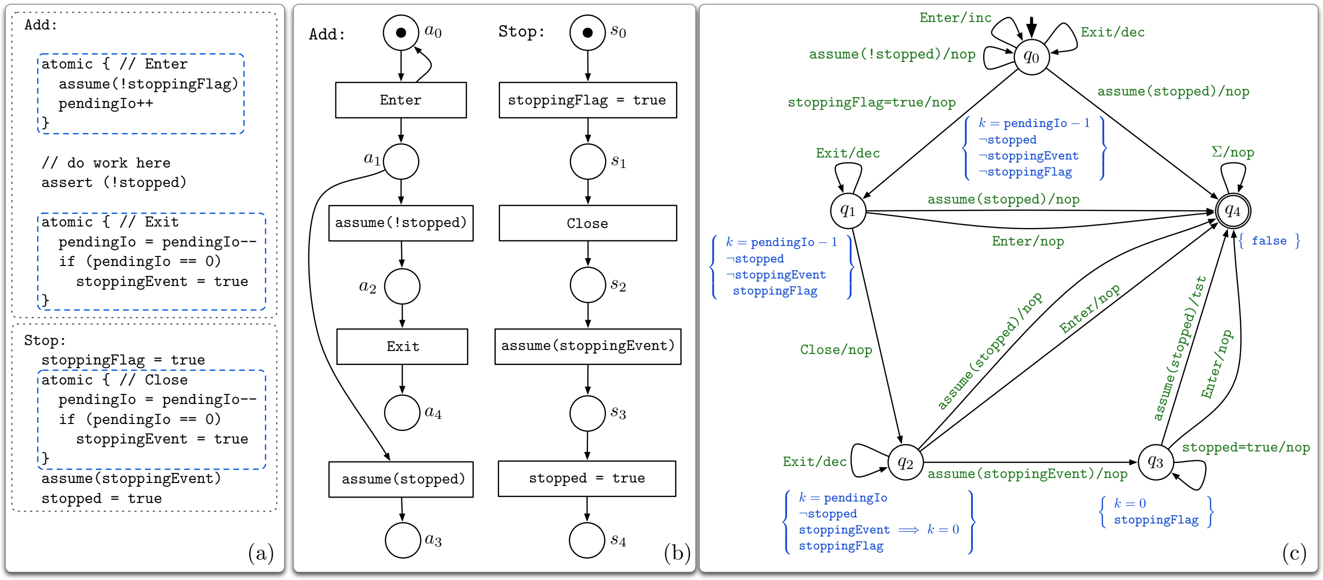
Our solution to (2) is based on the observation that counting automata can be converted into a kind of labelled Petri nets. To enable our language inclusion checking procedure we use *control flow nets* as our program model. A control flow net is a hybrid of a control flow graph and a Petri net. As in a control flow graph, a transition is labelled with an imperative program statement (over infinite data domains). As in a Petri net, tokens can be used to model infinite control, such as (parametrized) concurrency. Once we have modelled a program with a control flow net, problem (2) reduces to a Petri net language inclusion problem which is known to be decidable.

2. Motivating Example

We will formalize the verification problem for an industrially motivated example [46] through a control flow net and then present a counting proof.

Figure 3(a) presents a simplified version of the bluetooth device driver code (similar to the one that appears in [46]). The program consists of an arbitrary number of *work threads*, which all execute `Add`, and one *stop thread*, which executes `Stop`. The global variables are the integer variable `pendingIo` (initially 1), and the Boolean flags `stopped`, `stoppingEvent`, and `stoppingFlag`, (initially *false*).

We want to show that the statement `assert(!stopped)` never fails, which means that no configuration is reachable where one of the work threads is still working after the stop thread has ex-



```
Add:
  atomic { // Enter
    assume(!stoppingFlag)
    pendingIo++
  }

  // do work here
  assert (!stopped)

  atomic { // Exit
    pendingIo = pendingIo--
    if (pendingIo == 0)
      stoppingEvent = true
  }
Stop:
  stoppingFlag = true
  atomic { // Close
    pendingIo = pendingIo--
    if (pendingIo == 0)
      stoppingEvent = true
  }
  assume(stoppingEvent)
  stopped = true
```

Figure 3. (a) Bluetooth device driver code. (b) Control flow net for Bluetooth. The initial marking is as shown (no token other than on a_0 and on s_0). The final markings are those where at least a_3 has at least one token. (c) Bluetooth counting proof.

ecuted the statement `stopped = true`. The correctness relies on the atomicity of the three statements `Enter`, `Exit`, and `Close`. The bug detected in the version from [46] has been fixed by making `Enter` atomic.

Control Flow Net. The control flow net in Figure 3(b) models that an arbitrary number of work threads and one stop threads run in parallel. To model the fact that unboundedly many work threads can be active simultaneously, the initial `Add` transition `Enter` puts a token back on a_0, so that `Enter` remains enabled (just as if yet another work thread was spawned). The number of tokens on a_i models the number of work threads in the corresponding control location (there is always one token in `Stop` half of the net, since there is only one stop thread). A *final marking* is any marking where there is at least one token on the error location a_3. Each sequence of transitions (a *firing sequence*) of the net that reaches such a final marking corresponds to a thread interleaving that violates the assertion. The goal is to prove that the corresponding sequence of statements is infeasible (has the postcondition *false*).

Counting Proof. The counting automaton in Figure 3(c) is a finite state automaton augmented with a counter k. Transitions are labelled by letters of the alphabet of program statements and corresponding counter actions (the same letters are used to label the transitions in the control flow net). The counting proof consists of this automaton and the inductive annotation which maps the automaton states to assertions (blue terms in the figure).

Intuitively, at any moment while reading a trace τ (i.e., a sequence of statements), the value of the counter k records by how much the `Enter` transitions outnumber the `Exit` transitions (initially by 0).

The automaton states q_0, …, q_3 records what *statements of the stop thread* have appeared so far, in sequence: none (recorded by q_0), `stoppingFlag = true` (transition from q_0 into q_1), `Close` (transition from q_1 into q_2), `assume(stoppingEvent)` (transition from q_2 into q_3).

The automaton state q_4 is reached if: (1) the statement `assume(stopped)` of the work thread has appeared, or (2) the statement `Enter` of the work thread has appeared after the statement `stoppingFlag = true` of the stop thread has appeared.

It is easy to see that the annotation is inductive. Initially, the flags `stopped`, `stoppingEvent`, and `stoppingFlag` have the value *false*, `pendingIo` is 1, and k is 0, and thus, the assertion for the initial state q_0 is satisfied. One can check that for every transition, the corresponding Hoare triple holds.

The check of the trace inclusion between a control flow net and a counter automaton is decidable (as we will show in Section 6). To obtain an intuition why the inclusion here holds, consider the (informal) invariants: $s_0 = q_0$, $s_1 = q_1$, $s_2 = q_2$, $s_3 + s_4 = q_3$, and $a_1 + a_2 = k$.

The annotation of the final state q_4 with the assertion *false* means that every trace accepted by the counter automaton has postcondition *false*. Since every trace of the control flow net is accepted by the counter automaton, this means that the execution of the statement `assume(stopped)` is never feasible, i.e., the statement `assert(!stopped)` never fails, which is what we wanted to show.

3. Proofs That Count

One of the key contributions of this paper is a new formalism for proving program correctness, which we call *counting proofs*. A counting proof consists of a *counting automaton* and an *inductive annotation*. The automaton defines a set of traces and determines *what to count* (by associating counter actions to transitions in the automaton). The inductive annotation is a "correctness proof" for the counting automaton, which shows that every trace accepted by the automaton satisfies a given specification. Thus, we may prove the correctness of a program $\mathcal{P}$ by exhibiting a counting proof such that all traces of $\mathcal{P}$ belong to the counting automaton associated with the proof. In this section, we give a formal definition of our notion of counting proof.

Preliminaries

First, we define some terminology for our program model. We fix a (possibly infinite) set S of *memory states* (usually defined as valuations assigning values to a finite set of global and local variables), a set Σ of *program statements*, and a *semantic function* $[\![\cdot]\!] : \Sigma \to 2^{S \times S}$ which interprets each program statement as a relation between memory states. A *trace* is a sequence of statements (i.e., a word in Σ^*). Intuitively, we will think of a program as a set of traces; this allows us to abstract away from the control structure of the program, which may be complicated.

We fix a set of *program assertions*, which is denoted Φ, as well as an *entailment relation* $\models\ \subseteq S \times \Phi$. For $s \in S$ and $\varphi \in \Phi$, $s \models \varphi$ indicates that the assertion φ holds in state s. A (*pre/post*) *specification* for $\mathcal{P}$ consists of a pair $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}} \in \Phi$ indicating a *precondition* and *postcondition*. In the following, it will frequently be convenient to consider program assertions over an extended vocabulary that includes a fresh set of $\mathbb{N}$-sorted variable symbols V (depending on the application at hand, such a variable symbol may be interpreted as the values of some counter variables, or as the number of tokens on a place of a Petri net (Section 7)). The set of such *extended program assertions* will be denoted Φ^V.

For a statement $\sigma \in \Sigma$ and assertions φ, φ', we use the typical Hoare notation $\{\varphi\}\ \sigma\ \{\varphi'\}$ to denote that for every s, s' such that $s \models \varphi$ and $\langle s, s'\rangle \in [\![\sigma]\!]$, we have $s' \models \varphi'$. We extend this notation to sequences of statements in the obvious way.

Counting proofs

We may now introduce the definition of counting proofs. We start by defining counting automata, which are finite automata augmented with zero or more counter variables which can be incremented, decremented, and tested.

Definition 3.1 (Counting automaton) A (deterministic) *counting automaton* is a 6-tuple $A = \langle Q, n, q_0, \mathbf{k}_0, \delta, \Omega\rangle$ where Q is a finite set of states, $n \in \mathbb{N}$ is the *dimension* of the automaton (i.e., the number of counters used by the automaton), $q_0 \in Q$ is an initial state, $\mathbf{k}_0 \in \mathbb{N}^n$ is an initial vector, $\Omega : Q \to 2^{\mathbb{N}^n}$ is a mapping that takes each state to an upwards closed[1] set of final (accepting) vectors, and $\delta : Q \times \Sigma \rightharpoonup Q \times \mathsf{Act}^n$ is a partial function which maps (state, program statement) pairs to pairs consisting of a successor state and an action for each counter, where $\mathsf{Act} = \{\mathsf{inc}, \mathsf{dec}, \mathsf{nop}, \mathsf{tst}\}$. Note that since δ is partial, not every state is associated with a transition for each program statement. ⌟

Remark 3.2 A special case of particular interest is when the acceptance condition of a counting automaton $A = \langle Q, n, q_0, \mathbf{k}_0, \delta, \Omega\rangle$ is specified by a set of final states $F \subseteq Q$ (rather than a mapping Ω from states to upwards-closed sets of final vectors). This case is accommodated by setting $\Omega(q) = \mathbb{N}^n$ for every final state $q \in F$ and $\Omega(q') = \emptyset$ for every non-final state $q' \notin F$. This special case is used frequently in this paper, and in diagrams of counting automata we will use double circles to denote final states.

We also remark that nondeterministic finite automata are a special case of counting automata, where the dimension of the counting automaton is 0. Note that $\mathbb{N}^0$ is a singleton set, and so for any state q, the only choices for $\Omega(q)$ are $\mathbb{N}^0$ and $\emptyset$ (i.e., 0-dimensional counting automata always fall into the case described above, where the acceptance condition is determined by a set of final states). ⌟

We now introduce some additional terminology for counting automata. Let $A = \langle Q, n, q_0, \mathbf{k}_0, \delta, \Omega\rangle$ be a counting automaton. A *configuration* of A consists of a pair $\langle q, \mathbf{k}\rangle$, where $q \in Q$ is a state and $\mathbf{k} \in \mathbb{N}^n$ is a valuation for each counter. The *initial configuration* is $\langle q_0, \mathbf{k}_0\rangle$, and any configuration $\langle q, \mathbf{k}\rangle$ where $\mathbf{k} \in \Omega(q)$ is a *final configuration*. We lift δ to a partial transition function on configurations as follows:

$$\delta(\langle q, (k_1, k_2, ..., k_n)\rangle, c) \triangleq \langle q', (k_1', k_2', ..., k_n')\rangle$$

if $\delta(q, (k_1, k_2, ..., k_n)) = \langle q', (\alpha_1, \alpha_2, ..., \alpha_n)\rangle$
and each k_i' is defined as

$$k_i' \triangleq \begin{cases} k_i + 1 & \text{if } \alpha_i = \mathsf{inc} \\ k_i - 1 & \text{if } \alpha_i = \mathsf{dec} \land k_i > 0 \\ k_i & \text{if } \alpha_i = \mathsf{nop} \\ k_i & \text{if } \alpha_i = \mathsf{tst} \land k_i > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

We lift δ to a partial transition function δ^* on statement sequences in the obvious way. A trace $w \in \Sigma^*$ is *accepted* by A if $\delta^*(\langle q_0, \mathbf{k}_0\rangle, w)$ is a final configuration. The set of all traces accepted by A is denoted $\mathcal{L}(A)$.

We may now move on to our definition of an inductive annotation. In essence, an inductive annotation is a Floyd/Hoare proof.

Definition 3.3 (Annotation) Let $A = \langle Q, n, q_0, \mathbf{k}_0, \delta, \Omega\rangle$ be a counting automaton. Let $\mathtt{K} = \{\mathtt{k}_1, ..., \mathtt{k}_n\}$ be a set of distinguished $\mathbb{N}$-sorted variable symbols. An *annotation* is a map $\varphi : Q \to \Phi^{\mathtt{K}}$. We say φ is *inductive* if for all $q, q' \in Q$, $\sigma \in \Sigma$, and $\mathbf{k}, \mathbf{k}' \in \mathbb{N}^n$ such that $\delta(\langle q, \mathbf{k}\rangle, \sigma) = \langle q', \mathbf{k}'\rangle$ the Hoare triple

$$\{\varphi(q)[\mathbf{k}]\}\ \sigma\ \{\varphi(q')[\mathbf{k}']\}$$

holds (where for an assertion ψ and vector $\mathbf{k} = (k_1, ..., k_n) \in \mathbb{N}^n$, $\psi[\mathbf{k}]$ denotes the formula obtained by replacing each $\mathtt{k}_i$ with the corresponding element k_i of the vector $\mathbf{k}$). ⌟

Finally, we may define a counting proof as a pair consisting of a counting automaton and an inductive annotation for it.

Definition 3.4 (Counting proof) A *counting proof* is a pair $\langle A, \varphi\rangle$ consisting of a counting automaton $A = \langle Q, n, q_0, \mathbf{k}_0, \delta, \Omega\rangle$ and an inductive annotation φ for A.

We say that $\langle A, \varphi\rangle$ *satisfies* a given specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$ if the following two conditions hold:

(i) $\psi_{\mathsf{pre}} \models \varphi(q_0)[\mathbf{k}_0]$
(ii) For all $q \in Q$ and all $\mathbf{k}_f \in \Omega(q)$, we have $\varphi(q)[\mathbf{k}_f] \models \psi_{\mathsf{post}}$.

⌟

We now justify using counting proofs as proof objects. Recall that our use of counting proofs is the following (informal) proof rule: *Suppose that $\mathcal{P}$ is a program, $\langle A, \varphi\rangle$ is a counting proof which satisfies the specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$, and that every trace of $\mathcal{P}$ is accepted by A. Then $\mathcal{P}$ satisfies the specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$.* For this proof rule to be sound, we must show that every trace in $\mathcal{L}(A)$ satisfies the specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$. With this goal in mind, we first prove a lemma.

Lemma 3.5 Let $A = \langle Q, n, q_0, \mathbf{k}_0, \delta, \Omega\rangle$ be a counting automaton, and let φ be an inductive annotation for A. Let $\tau \in \Sigma^*$. If $q, q', \mathbf{k}, \mathbf{k}'$ are such that $\delta^*(\langle q, \mathbf{k}\rangle, \tau) = \langle q', \mathbf{k}'\rangle$, then the following Hoare triple holds:

$$\{\varphi(q)[\mathbf{k}]\}\ \tau\ \{\varphi(q')[\mathbf{k}']\}$$

⌟

Proof. By induction on τ. The base case (τ is empty) is trivial. The induction step follows from Definition 3.3 and the sequential composition rule for Hoare logic.

□

We may now state the main result of this section: a soundness theorem which justifies our proof rule.

Theorem 3.6 Let $\langle A, \varphi\rangle$ be a counting proof with $A = \langle Q, n, q_0, \mathbf{k}_0, \delta, \Omega\rangle$, and let ψ_{pre} and ψ_{post} be assertions such that

[1] Recall that a set of vectors V is upwards closed if for all $\mathbf{v} \in V$ and all $\mathbf{v}'$ such that $\mathbf{v} \leq \mathbf{v}'$, we have $\mathbf{v}' \in V$.

$\langle A, \varphi \rangle$ satisfies $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$. Then for any $\tau \in \mathcal{L}(A)$, τ satisfies the Hoare triple $\{\psi_{\mathsf{pre}}\}\ \tau\ \{\psi_{\mathsf{post}}\}$. ⌟

Proof. Follows from Lemma 3.5, Definition 3.4, the definition of $\mathcal{L}(A)$, and the consequence rule of Hoare logic. □

4. Constructing Counting Proofs

We now consider the problem of automatically synthesizing a counting proof which certifies that a given (finite) set of traces $Tr \subseteq \Sigma^*$ is correct with respect to a given specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$. Our algorithm for constructing counting proofs is divided into two parts. The first part is a decision procedure which determines whether a counting proof of a given "size" exists for a given set of traces and specification (here "size" should be understood informally). The second part is a search procedure, which repeatedly calls the decision procedure on different candidate sizes until a proof is found, and attempts to find the smallest proof possible. We start by describing the decision procedure, and then discuss the search procedure.

The idea behind our proof construction procedure is to encode the proof construction problem as a formula Ψ and use an off-the-shelf SMT solver to find a model for Ψ. From any such model, we may extract a counting proof $\langle A, \varphi \rangle$ which satisfies $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$ and such that $Tr \subseteq \mathcal{L}(A)$.

Since the goal is to develop a decision procedure for proof construction, we must limit the program statement language and the language of program assertions to a decidable logic. A *convex linear real formula* is a conjunction of a finite number of linear inequalities over a set of real variables with real coefficients. The number of inequalities appearing in a convex linear real formula ψ will be called the *size* of ψ, and denoted $|\psi|$.

In this section we fix a set of traces Tr and a pre/post specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$. We will assume that ψ_{pre} and ψ_{post} are convex linear real formulae, and that every program statement can be expressed as a convex linear real formula. For simplicity, we will also assume that Tr consists of a single trace $\tau = \sigma_1 \cdots \sigma_{|\tau|}$ (the generalization to multiple traces is straightforward).

Let $\langle A, \varphi \rangle$ be a counting proof, with $A = \langle Q, n, q_0, \mathbf{k}_0, \delta, \Omega \rangle$. The "size" of $\langle A, \varphi \rangle$ is determined by four numbers:

1. The number of states, $\#\mathsf{States}(A, \varphi) \triangleq |Q|$
2. The dimension of A, $\mathsf{dim}(A, \varphi) \triangleq n$
3. The size of the assertions used in the annotation φ,
$$\#\mathsf{Atoms}(A, \varphi) \triangleq \sum_{q \in Q} |\varphi(q)|$$
4. The number of minimal final vectors:
$$\#\mathsf{Min}(A, \varphi) \triangleq \sum_{q \in Q} |\{\mathbf{v} : \nexists \mathbf{v}' \in \Omega(q).\mathbf{v}' < \mathbf{v}\}|$$

Our decision procedure will take four additional parameters as input $\overline{Q}, \overline{N}, \overline{K}, \overline{M} \in \mathbb{N}$, corresponding (in order) to the four components of counting proof size described above. The idea behind this algorithm is to construct a QF_UFNRA[2] formula which is satisfiable iff there exists a counting proof $\langle A, \varphi \rangle$ such that A accepts τ, A satisfies $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$, $\#\mathsf{States}(A, \varphi) = \overline{Q}$, $\mathsf{dim}(A, \varphi) = \overline{N}$, $\#\mathsf{Assert}(A, \varphi) = \overline{K}$, and $\#\mathsf{Min}(A, \varphi) = \overline{M}$.

For presentation purposes, we assume that $\overline{N} = \overline{M} = 1$ and that we are searching for a proof with $\overline{K} = \overline{Q}$ with one atom per state (but that $\overline{Q}$ is arbitrary). The generalization of the construction to arbitrary values of the parameters $\overline{N}$, $\overline{M}$, and $\overline{K}$ is straightforward. The construction of the QF_UFNRA formula follows.

Automaton constraints Without loss of generality, we may construct a counting proof with state space $Q = \{1, ..., \overline{Q}\}$. Recall that the transition function of a 1-dimensional counting automaton over the state space Q is a partial function

$$\delta : Q \times \Sigma \rightharpoonup Q \times \mathsf{Act},$$

where $\mathsf{Act} = \{\mathsf{inc}, \mathsf{dec}, \mathsf{nop}, \mathsf{tst}\}$ is a set of counter actions. For our encoding, we introduce for each $\sigma \in \Sigma(\tau) \triangleq \{\sigma_1, ..., \sigma_{|\tau|}\}$ a pair of uninterpreted function symbols

$$\delta_\sigma : Q \to Q$$
$$\alpha_\sigma : Q \to \mathsf{Act}$$

We now encode the constraint that the counting automaton accepts the trace τ. We introduce a set of integer variables $q_0, ..., q_{|\tau|}, k_0, ..., k_{|\tau|}$; the interpretation of these variables is that, after reading the prefix $\sigma_1\sigma_2 \cdots \sigma_i$ of the trace τ, the state of the counter automaton is q_i, and the value of the counter is k_i (q_0 and k_0 represent the initial state and initial value of the counter). This interpretation is encoded by the following constraints:

$$\Psi_{\delta\text{-}\mathsf{state}} \triangleq \bigwedge_{i=1}^{|\tau|} \delta_{\sigma_i}(q_{i-1}) = q_i$$

$$\Psi_{\delta\text{-}\mathsf{counter}} \triangleq k_0 \geq 0 \wedge \bigwedge_{i=1}^{|\tau|} k_i = k_{i-1} + \mathsf{incr}(\alpha_{\sigma_i}(q_{i-1}))$$

where

$$\mathsf{incr}(\alpha) \triangleq \begin{cases} 1 & \text{if } \alpha = \mathsf{inc} \\ -1 & \text{if } \alpha = \mathsf{dec} \\ 0 & \text{otherwise} \end{cases}$$

The preceding constraints ensure that τ labels a path through the counting automaton and that at each step along this path k_i is updated appropriately. Another constraint is required for τ to be accepted by the counting automaton, which is that none of the transitions that label the accepting path may block (recall that the dec and tst counter actions block when the counter is 0). This non-blocking condition is enforced by the following constraint:

$$\Psi_{\delta\text{-}\mathsf{block}} \triangleq \bigwedge_{i=1}^{|\tau|} \Big(\alpha_{\sigma_i}(q_{i-1}) \in \{\mathsf{dec}, \mathsf{tst}\} \Rightarrow k_{i-1} \geq 1 \Big)$$

The last requirement to ensure that τ is accepted by the counting automaton is that we must synthesize an upwards closed set of final vectors for the state $q_{|\tau|}$. For the case of 1-dimensional counting automata, an upwards closed set is completely determined by a single natural number, which is the lower bound on the set. Thus, we introduce an integer variable k_{final} and create a constraint that the value of the counter after reading τ is greater than or equal to k_{final}:

$$\Psi_{\mathsf{accept}} \triangleq k_{\mathit{final}} \leq k_{|\tau|}$$

Collecting these constraints, we have

$$\Psi_A \triangleq \Psi_{\delta\text{-}\mathsf{state}} \wedge \Psi_{\delta\text{-}\mathsf{counter}} \wedge \Psi_{\delta\text{-}\mathsf{block}} \wedge \Psi_{\mathsf{accept}}$$

Models of Ψ_A correspond to $\overline{Q}$-state, 1-dimensional counting automata which accepts the trace τ.

Annotation constraints We now consider the constraints used to construct the inductive annotation of the counting automaton. We use a constraint-based technique to synthesize linear invariants, as described in [15].

[2] Quantifier-free non-linear real arithmetic formulae with uninterpreted function symbols

Fix a set of program variables $X = \{x_1, ..., x_{|X|}\}$. For each program variable x_i, we introduce an uninterpreted function symbol C_{x_i} of sort $Q \to \mathbb{R}$. We also introduce two additional uninterpreted function symbols C_k, C_1 of sort $Q \to \mathbb{R}$. For a given state $q \in Q$, $C_{x_i}(q)$ will be the coefficient of x_i, $C_{\mathtt{k}}(q)$ will be the coefficient of $\mathtt{k}$, $C_1(q)$ will be the constant coefficient in the linear inequality assigned to q. That is, for $q \in Q$, the annotation at q will be:

$$\mathsf{inv}(q) \triangleq \Big(\sum_{i=1}^{|X|} C_{x_i}(q) \cdot x_i\Big) + C_{\mathtt{k}}(q) \cdot \mathtt{k} + C_1(q) \leq 0$$

For any statement $\sigma \in \Sigma$, we use $\mathsf{tf}(\sigma)$ to denote the *transition formula* of σ, which is a formula over the variables $x_1, ..., x_{|X|}$ and their "primed" copies $x'_1, ..., x'_{|X|}$ that represents the meaning of the statement σ. For a counter machine action $\alpha \in \mathsf{Act}$, we use $\mathsf{tf}(\alpha)$ to denote the transition formula corresponding to α, defined below as:

$$\mathsf{tf}(\alpha) \triangleq \begin{cases} \mathtt{k}' = \mathtt{k} + 1 & \text{if } \alpha = \mathsf{inc} \\ \mathtt{k} \geq 1 \wedge \mathtt{k}' = \mathtt{k} - 1 & \text{if } \alpha = \mathsf{dec} \\ \mathtt{k} \geq 1 \wedge \mathtt{k}' = \mathtt{k} & \text{if } \alpha = \mathsf{tst} \\ \mathtt{k}' = \mathtt{k} & \text{if } \alpha = \mathsf{nop} \end{cases}$$

We may now define the consecution constraints, which ensure that the annotation is inductive. We define a consecution constraint for each $i \in \{1, ..., |\tau|\}$ as follows:

$$\Psi_i \triangleq (\forall \mathtt{k}, \mathtt{k}', X, X')(\mathsf{inv}(q_{i-1}) \wedge \mathsf{tf}(\sigma_i) \wedge \mathsf{tf}(\alpha_{\sigma_i}(q_{i-1})) \Rightarrow \mathsf{inv}(q_i)')$$

where $\mathsf{inv}(q_i)'$ denotes the formula obtained from $\mathsf{inv}(q_i)$ by replacing each variable x_i with its primed copy x'_i and k with k'.

The last annotation constraints ensure that the counting proof satisfies the specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$:

$$\Psi_{\mathsf{init}} \triangleq (\forall \mathtt{k}, X)(\psi_{\mathsf{pre}} \Rightarrow \mathsf{inv}(q_0))$$

$$\Psi_{\mathsf{final}} \triangleq (\forall \mathtt{k}, X)(\mathsf{inv}(q_{|\tau|}) \wedge \mathtt{k} \geq k_{\mathit{final}} \Rightarrow \psi_{\mathsf{post}})$$

The annotation constraints defined in this section all feature universal quantification. However, all the formulae defined in this section belong to a class of formulae for which the universal quantifiers can be replaced by existential quantifiers by applying Farkas' lemma (see [15] for details). We will use $\Psi_i^{\exists}$, $\Psi_{\mathsf{init}}^{\exists}$ and $\Psi_{\mathsf{final}}^{\exists}$ to denote the corresponding transformed formulae.

Finally, we collect the annotation constraints into a single formula Ψ_φ:

$$\Psi_\varphi \triangleq \Psi_{\mathsf{init}}^{\exists} \wedge \Psi_{\mathsf{final}}^{\exists} \wedge \Big(\bigwedge_{i=1}^{|\tau|} \Psi_i^{\exists}\Big)$$

Counting proof extraction We conjoin the automaton and annotation constraints to arrive at a formula Ψ whose models correspond to counting proofs:

$$\Psi \triangleq \Psi_A \wedge \Psi_\varphi$$

Suppose that $\mathcal{M}$ is a model of Ψ. For a term *term*, we use $\mathit{term}^{\mathcal{M}}$ to denote the interpretation of *term* in the model $\mathcal{M}$. A counting proof can be constructed as follows:

$$\begin{aligned} Q &\triangleq \{1, ..., \overline{Q}\} \\ q_0 &\triangleq q_0^{\mathcal{M}} \\ \mathbf{k}_0 &\triangleq k_0^{\mathcal{M}} \\ \Omega &\triangleq \lambda q. \begin{cases} \{k : k \geq k_{\mathit{final}}\} & \text{if } q = q_{n+1}^{\mathcal{M}} \\ \emptyset & \text{otherwise} \end{cases} \\ \delta &\triangleq \lambda\langle q, \sigma\rangle. \begin{cases} \langle \delta_\sigma(q)^{\mathcal{M}}, \alpha_\sigma(q)^{\mathcal{M}}\rangle & \text{if } \exists i. q = q_i \wedge \sigma = \sigma_{i-1} \\ \text{undefined} & \text{otherwise} \end{cases} \\ \varphi &\triangleq \lambda q. \Big(\sum_{i=1}^{|X|} C_{x_i}(q)^{\mathcal{M}} \cdot x_i\Big) + C_{\mathtt{k}}(q)^{\mathcal{M}} \cdot \mathtt{k} + C_1(q)^{\mathcal{M}} \leq 0 \end{aligned}$$

The following proposition states that our encoding is, in a sense, complete.

Proposition 4.1 Let *Tr* be a finite set of traces (such that each command is expressible as a convex linear real formula) and let $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$ be a (convex linear real) specification. Let $\overline{N}, \overline{Q}, \overline{K}, \overline{M} \in \mathbb{N}$. The problem of *determining whether there exists a counting proof $\langle A, \varphi\rangle$ that satisfies the specification ψ_{pre}/ψ_{post}, such that $Tr \subseteq \mathcal{L}(A)$, and such that $\#\mathsf{States}(A, \varphi) = \overline{Q}$, $\mathsf{dim}(A, \varphi) = \overline{N}$, $\#\mathsf{Assert}(A, \varphi) = \overline{K}$, and $\#\mathsf{Min}(A, \varphi) = \overline{M}$*, is decidable. ⌟

4.1 Searching for Small Counting Proofs

The decision procedure presented in the preceding section suggests a simple algorithm for constructing a counting proof. Suppose we are given a set of traces *Tr* and a specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$. First, we check whether some $\tau \in \mathit{Tr}$ violates the specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$ (this can be accomplished with an SMT query). If such a τ exists, we are done: no proof exists. Otherwise, enumerate the space of parameters $\langle \overline{N}, \overline{Q}, \overline{K}, \overline{M}\rangle \in \mathbb{N}^4$, applying the decision procedure described in the previous section until a counting proof is found.

The termination of this simple algorithm relies on the following observation:

Observation 4.2 Let *Tr* be a set of traces and let $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$ be a specification such that for each $\tau \in \mathit{Tr}$, $\{\psi_{\mathsf{pre}}\}\ \tau\ \{\psi_{\mathsf{post}}\}$. Then there is a counting proof $\langle A, \varphi\rangle$ which satisfies $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$ and which accepts every trace in *Tr*. ⌟

The intuition behind this observation is that we can always find a counting proof where the counting automaton is 0-dimensional (i.e., a counting automaton that does not use counters) and shaped like a prefix tree. The existence of an inductive annotation for such an automaton is a consequence of Farkas' lemma.

While this observation is enough to prove termination of our proof synthesis procedure, it is not very satisfying: it says that in the worst case, we can always construct a counting proof that does not, in fact, count. The key insight behind our counting proof synthesis procedure is that we can constrain the parameters given to the decision problem to force an SMT solver to discover a non-trivial counting argument. For example, consider the sample trace τ from Section 1. Setting $\overline{Q} = 2, \overline{K} = 2, \overline{N} = \overline{M} = 1$, we can force the SMT to synthesize a non-trivial counter, simply because there is no 0-dimensional counting proof for these traces that only uses 2 states (so, by virtue of the fact that our decision procedure *will* find a 2-state proof if one exists, it must find a proof that uses a non-trivial counting argument).

In the remainder of this section, we present a refinement on the simple algorithm discussed above, which fixes a particular way of enumerating the parameter space. The intuition behind this refinement is that we want to synthesize a counting proof that minimizes the sum of the number of states ($\overline{Q}$) and the number of assertions ($\overline{K}$).[3] Our aim in presenting this algorithm is not to present a practical proof construction procedure, but rather a theoretical algorithm capable of synthesizing proofs that are (in a sense) optimal. In practice, one would expect a heuristic-based method for exploring the

[3] This algorithm can be adapted to minimize any objective function which is strictly increasing in $\overline{Q}$ and $\overline{K}$.

parameter space (which cannot necessarily guarantee optimality) would be more effective.

Observation 4.2 yields a coarse upper bound on the sum of the number of states and atoms used in a counting proof for a given set of traces *Tr* of $2 \cdot \sum_{\tau \in Tr} |\tau|$. Since we have an upper bound on $\overline{Q} + \overline{K}$, the parameter space $\overline{Q}, \overline{K} \in \mathbb{N} \times \mathbb{N}$ can be searched efficiently using binary search. The termination of this algorithm relies on the following proposition:

Proposition 4.3 Let *Tr* be a finite set of traces, $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$ be a specification, and $Z \in \mathbb{N}$. The problem of determining whether there exists a counting proof $\langle A, \varphi \rangle$ that satisfies $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$ such that $Tr \subseteq \mathcal{L}(A)$, and such that $\#\mathsf{Atoms}(A, \varphi) + \#\mathsf{States}(A, \varphi) = Z$ is decidable. $\lrcorner$

This proposition is not trivial, because even though fixing the size of $\#\mathsf{Atoms}(A, \varphi) + \#\mathsf{States}(A, \varphi)$ implies finitely many choices for $\overline{Q}$ and $\overline{K}$, the space for the remaining parameters $\overline{N}$ and $\overline{M}$ is infinite and cannot be searched exhaustively. The following two lemmas imply that we may place a finite upper bound on $\overline{N}$ and $\overline{M}$ once *Tr* and $\overline{Q}$ are fixed.

Lemma 4.4 Let $A = \langle Q, n, q_0, \mathbf{k}_0, \delta, \Omega \rangle$ be a counting automaton, and let Tr be a finite set of traces which are accepted by A. There exists a mapping $\Omega' : Q \to 2^{\mathbb{N}^n}$ such that the counting proof $\langle A', \varphi \rangle$ with $A' = \langle Q, n, q_0, \mathbf{k}_0, \delta, \Omega' \rangle$ satisfies

$$\#\mathsf{Min}(A', \varphi) \leq |Tr|$$

and $Tr \subseteq \mathcal{L}(A')$. $\lrcorner$

Proof. Intuitively, we define Ω' to be the upwards closure of the set of configurations that result from reading the traces in Tr. Formally, for any $q \in Q$, define

$$\Omega'(q) \triangleq \{\mathbf{k}' \in \mathbb{N}^n : (\exists \tau \in Tr, \exists \mathbf{k} \in \mathbb{N}^n)(\mathbf{k} \leq \mathbf{k}' \wedge \delta^*(\langle q_0, \mathbf{k}_0 \rangle, \tau) = \langle q, \mathbf{k} \rangle)\}$$

□

Lemma 4.5 Let $\langle \varphi, A \rangle$ be a counting proof, with $A = \langle Q, n, q_0, \mathbf{k}_0, \delta, \Omega \rangle$. There exists a counting proof $\langle \varphi', A' \rangle$ such that A' recognizes the same language as A, and A' has dimension at most $3^{|\mathsf{dom}(\delta)|}$ (where $|\mathsf{dom}(\delta)|$ indicates the cardinality of the domain of the partial transition function δ). $\lrcorner$

Proof. Suppose that $n > 3^{|\mathsf{dom}(\delta)|}$. By the pigeonhole principle, there exists distinct counters $s, t \in \{1, ..., n\}$ which agree on all their actions. That is, for any $q \in Q, \sigma \in \Sigma$, if $\delta(q, \sigma) = (\alpha_1, ..., \alpha_n)$, then $\alpha_s = \alpha_t$. Without loss of generality, we can assume that $s = 1$ and $t = 2$.

Letting $(k_1, ..., k_n) = \mathbf{k}_0$, we define $\Delta \triangleq k_1 - k_2$. Without loss of generality, we may assume that $\Delta \geq 0$. We construct an $(n-1)$-dimensional counting automaton $A' = \langle Q, n-1, q_0, \mathbf{k}_0', \delta', \Omega' \rangle$ and an inductive annotation φ' as follows:

- $\mathbf{k}_0' \triangleq (k_2, ..., k_n)$, where $(k_1, ..., k_n) = \mathbf{k}_0$
- For any q, σ such that $\delta(q, \sigma) = \langle q', (\alpha_1, ..., \alpha_n) \rangle$, we define

  $$\delta'(q, \sigma) \triangleq \langle q', (\alpha_2, ..., \alpha_n) \rangle$$
- For any $q \in Q$, we define

  $$\Omega'(q) \triangleq \{(\min\{k_1 - \Delta, k_2\}, k_3, ..., k_n) : (k_1, ..., k_n) \in \Omega(q)\}$$
- For any $q \in Q$, we define

  $$\varphi'(q) \triangleq (\exists \mathtt{k}_1)(\mathtt{k}_2 - \Delta = \mathtt{k}_1 \wedge \varphi(q))$$

  (the quantifier $(\exists \mathtt{k}_1)$ can be eliminated to yield a convex linear real formula). □

The proof of Proposition 4.3 follows easily from the preceding two lemmata.

We conclude this section with an algorithm summarizing the search procedure described above. In this algorithm, we use $\mathsf{constraints}(...)$ to denote the formula constructed in the previous section, and $\mathsf{extract\text{-}proof})(\Psi)$ to denote the counting proof extracted from a model of a formula Ψ.

```
Input: $Tr, \psi_{\mathsf{pre}}, \psi_{\mathsf{post}}$ such that for all $\tau \in Tr$,
       $\{\psi_{\mathsf{pre}}\}\ \tau\ \{\psi_{\mathsf{post}}\}$
Output: A counting proof $\langle A, \varphi \rangle$ which satisfies $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$
        and with $Tr \subseteq \mathcal{L}(A)$
$min \leftarrow 2$;
$max \leftarrow 2 \cdot \sum_{\tau \in | Tr} |\tau|$;
$\overline{M} = |Tr|$;
/* Binary search for minimal proof                  */
while $max \neq min$ do
    $mid \leftarrow (max + min)/2$;
    $\overline{Q} \leftarrow mid - 1$;
    $\overline{K} \leftarrow 1$;
    $\overline{N} \leftarrow 3^{|\Sigma(Tr)| \cdot \overline{Q}}$; // $\Sigma(Tr) = \bigcup\{\Sigma(\tau) : \tau \in Tr\}$
    $\Psi \leftarrow \mathsf{constraints}(Tr, \psi_{\mathsf{pre}}, \psi_{\mathsf{post}}, \overline{N}, \overline{Q}, \overline{K}, \overline{M})$;
    /* Search for proof of size $\overline{Q} + \overline{K}$          */
    while $\Psi$ is unsatisfiable and $\overline{Q} \geq 1$ do
        $(\overline{K}, \overline{Q}) \leftarrow (\overline{K} + 1, \overline{Q} - 1)$;
        $\overline{N} \leftarrow 3^{|\Sigma(Tr)| \cdot \overline{Q}}$;
        $\Psi \leftarrow \mathsf{constraints}(Tr, \psi_{\mathsf{pre}}, \psi_{\mathsf{post}}, \overline{N}, \overline{Q}, \overline{K}, \overline{M})$;
    end
    if $\Psi$ is satisfiable then
        $\langle A, \varphi \rangle \leftarrow \mathsf{extract\text{-}proof}(\Psi)$;
        $max \leftarrow mid$;
    end
    $min \leftarrow mid + 1$;
end
return $\langle A, \varphi \rangle$
```

Algorithm 1: Counting proof construction

5. Control Flow Nets

In the preceding, we have considered the control structure of a program to be a black box. In order to implement the algorithm outlined in Figure 1, we need an effective procedure for checking whether a given counting automaton accepts all the traces of a given program. In this section, we introduce *control flow nets*, an expressive program model that features infinite control, but for which the inclusion check is decidable (Section 6).

We begin by introducing some notions from Petri net theory, which form the basis of control flow nets.

Definition 5.1 (Petri net structure) A *Petri net structure* is a tuple $N = \langle P, T, \mathcal{E} \rangle$, where P is a finite set of *places*, T is a finite set of *transitions*, and $\mathcal{E} \subseteq (P \times T) \cup (T \times P)$ is an *incidence relation* connecting places to transitions and vice-versa. $\lrcorner$

We now recall some standard definitions for Petri nets. Let $N = \langle P, T, \mathcal{E} \rangle$ be a Petri net structure. Given a transition $t \in T$, we define its *pre-set*, ${}^\bullet t$, and *post-set*, $t^\bullet$, as the set of places with incoming/outgoing arcs to t:

$${}^\bullet t \triangleq \{p : \langle p, t \rangle \in \mathcal{E}\} \qquad t^\bullet \triangleq \{p : \langle t, p \rangle \in \mathcal{E}\}$$

A *marking* of N is a map $m : P \to \mathbb{N}$. Given a marking m and a transition $t \in T$, we say that t is *enabled in* m if for all $p \in {}^\bullet t$,

$m(p) \geq 1$. We write $m \xrightarrow{t} m'$ to denote that t is enabled in m and that for all p,

$$m'(p) = m(p) - |\{p\} \cap {}^\bullet t| + |\{p\} \cap t^\bullet|$$

For markings m, m' we write $m \preceq m'$ if for all p, $m(p) \leq m'(p)$. A set of markings M is *upwards closed* if for all $m \in M$ and all m' such that $m \preceq m'$, we have $m' \in M$. Any upwards closed set of markings can be represented by its set of minimal elements, which is always finite.

We may now introduce our program model, *control flow nets*. Control flow nets are programs which have Petri nets as control structures. We may think of them as an infinite analogue of control flow graphs: just as a control flow graph is a finite automaton labelled by program statements, a control flow net is a Petri net structure with transitions labelled by program statements. A slightly unusual feature (although not without precedent [25]) is that we also include acceptance conditions for control flow nets – this enables us to view control flow nets as language recognition devices. Formally,

Definition 5.2 (Control flow net) A *control flow net* is a tuple $\mathcal{P} = \langle P, T, \mathcal{E}, \ell, m_0, F \rangle$, where $\langle P, T, \mathcal{E} \rangle$ is a Petri net structure, $\ell : T \to \Sigma$ is an injective map which labels transitions with program statements, m_0 is an initial marking and F is an upwards closed set of final markings. ⌟

Control flow nets are a very expressive program model. For example, Figure 3(b) depicts a control flow net for a concurrent program with arbitrarily many threads. With their foundations on Petri nets, modelling concurrency is a particularly strong suit for control flow nets, but they can also be useful for representing other control features. For example, Figure 4 depicts a control flow net where tokens are used to count the number of stack frames in a recursive program.

In view of our black-box strategy for counting proof synthesis, we conclude this section with a definition of the traces and correctness of a control flow net.

Definition 5.3 (Correctness) A *trace* of a control flow net $\mathcal{P} = \langle P, T, \mathcal{E}, \ell, m_0, F \rangle$ is a sequence of statements $\tau = \sigma_1 ... \sigma_n \in \Sigma^*$ such that there exists transitions $t_1, ..., t_n$ where $\ell(t_1) = \sigma_1, \ldots, \ell(t_n) = \sigma_n$, and markings $m_1, ..., m_n$ where $m_n \in F$ such that

$$m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} m_n \ .$$

The set of all traces of $\mathcal{P}$ is denoted $\mathcal{L}(\mathcal{P})$. Given a pre/postcondition pair $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$, the control flow net $\mathcal{P}$ is *correct* if all of its traces are correct, i.e., if the Hoare triple $\{\psi_{\mathsf{pre}}\}\ \tau\ \{\psi_{\mathsf{post}}\}$ holds for all $\tau \in \mathcal{L}(\mathcal{P})$. ⌟

6. Proof Checking

A counting proof $\langle A, \varphi \rangle$ which satisfies a specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$ represents a set of correct behaviours which are correct with respect to that specification. If every trace of a given program $\mathcal{P}$ is represented by A, this implies that $\mathcal{P}$ is correct. Thus, in order to check a counting proof, we must check the inclusion of the language of traces of a control flow net $\mathcal{P}$ inside the language of a counting automaton. In this section, we prove that this check is decidable and give a characterization of the computational complexity of proof checking.

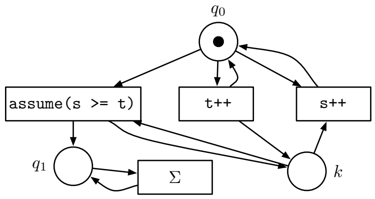
First, we state our decidability result. The essence of the proof of this theorem is that counting proof checking can be reduced to Petri net reachability, which is known to be decidable [35].

Theorem 6.1 Let $\mathcal{P} = \langle P, T, \mathcal{E}, \ell, m_0, F \rangle$ be a control flow net and let $A = \langle Q, n, q_0, \mathbf{k}_0, \delta, \Omega \rangle$ be a counting automaton. The problem of checking the inclusion $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(A)$ is decidable.

This result still holds even if we relax the accepting sets for control flow nets and/or counting automata to be semi-linear rather than upwards closed. ⌟

Proof. The proof proceeds by reduction to an inclusion problem for Petri net languages, a problem known to be reducible to Petri net reachability. It is sufficient to show that for any counting automaton we may construct a control flow net which recognizes the same language, which is a *deterministic* Petri net (by construction). From the point of view of language recognition, control flow nets are equivalent to L-type labelled Petri nets.[4] In [42] it is show that deterministic L-type Petri net languages are closed under compliment, and in [26], it is shown that the problem of checking the inclusion of an L-type Petri net language in a deterministic L-type Petri net language is decidable.

An example illustrating the construction of a control flow net from a counting automaton in the figure below, which depicts the control flow net associated with the counting automaton for the example in Section 1. Note that in this picture, the transition labelled Σ represents three (parallel) transitions, one for each letter in $\Sigma = \{$`t++`, `s++`, `assume(s >= t)`$\}$. In the rest of this proof, we make the construction formal.



Let $A = \langle Q, n, q_0, \mathbf{k}_0, \delta, \Omega \rangle$ be a counting automaton. We define a control flow net $\mathcal{P}_A = \langle P, T, \mathcal{E}, \ell, m_0, F \rangle$ as follows. The places of $\mathcal{P}_A$ are defined to be the disjoint union of the set of states of A and a set of n distinguished "counter" places $p_1, ..., p_n$:

$$P = Q + \{p_1, ..., p_n\}$$

The transitions correspond to points in the domain of δ:

$$T = \{\langle q, \sigma \rangle \in Q \times \Sigma : \delta(q, \sigma) \text{ is defined}\}$$

We define the incidence relation $\mathcal{E}$ indirectly, by defining the pre-set and post-set of every transition. Let $\langle q, \sigma \rangle \in T$ and suppose that $\delta(q, \sigma) = \langle q', (\alpha_1, ..., \alpha_n) \rangle$. Then we may define:

$$^\bullet\langle q, \sigma \rangle = \{q\} \cup \{p_i : \alpha_i = \mathsf{dec} \vee \alpha_i = \mathsf{tst}\}$$

$$\langle q, \sigma \rangle^\bullet = \{q'\} \cup \{p_i : \alpha_i = \mathsf{inc} \vee \alpha_i = \mathsf{tst}\}$$

We define a map ζ from configurations of A to markings of $\mathcal{P}_A$. For a given configuration $\langle q, (k_1, ..., k_n) \rangle$, we define

$$\zeta(q, (k_1, ..., k_n)) = \lambda p. \begin{cases} 1 & \text{if } p = q \\ k_i & \text{if } p = p_i \\ 0 & \text{otherwise} \end{cases}$$

We define the initial marking m_0 to be $\zeta(q_0, \mathbf{k}_0)$ and we define the (upwards closed) set of final markings F by taking the image of the accepting configurations of A under ζ. The labelling function ℓ is defined by $\ell(q, \sigma) = \sigma$.

It is easy to prove that this mapping ζ from configurations of A to markings of $\mathcal{P}_A$ is an isomorphism when restricted to the reachable configurations of A and the reachable markings of $\mathcal{P}_A$. Lastly, it is clear to see that $\delta(q, \mathbf{k}) = \langle q', \mathbf{k}' \rangle$ iff $\zeta(q, \mathbf{k}) \to \zeta(q', \mathbf{k}')$, which completes the proof of equivalence. □

[4] There is a slight technical difference: the labelling function for labelled Petri nets is not required to be injective. We will, in fact, ignore the injectivity restriction in our construction.

6.1 Complexity of Proof Checking

The decidability of this language inclusion problem is of fundamental importance, because it justifies calling counting proofs "proofs." A desirable result would be that checking proofs is not only decidable but *efficiently* so. Theorem 6.1 relies on a reduction to Petri net reachability. This problem is at least EXPSPACE-hard, but the best known upper bound for its complexity is NONELEMENTARY. So, a natural question is: *can we do better than reducing to Petri net reachability*? The following theorem states that we can not in general: Petri net reachability and counting proof checking are equivalent from the perspective of computational complexity.

Theorem 6.2 The Petri net reachability problem is polytime interreducible with the counting proof checking problem. $\lrcorner$

Proof. One direction of the proof is given in the proof of Theorem 6.1 (it is easy to prove that the reduction to reachability is polytime). For the other direction, we give a polytime reduction from the single-place zero-reachability problem to proof checking. This implies the result due to Lemma 4.1 in [24]. In fact, [24] states a slightly weaker result: that reachability is recursively reducible to single-place zero-reachability. However, it is easy to see that the reduction from the proof of [24] is polytime.

Consider a Petri net $G = \langle P, T, \mathcal{E} \rangle$, a marking m of G, and a place $p \in P$. Given an initial marking m_0, we say that a place p is zero-reachable if there exists a marking m' which is reachable from m_0 and such that $m'(p) = 0$. We reduce the problem of checking zero-reachability of p to counting proof checking by constructing a control flow net $\mathcal{P}$ and a counting automaton A such that A accepts all the traces of $\mathcal{P}$ iff p is zero-reachable.

We suppose that zr is a new transition which does not belong to T. We take $\mathcal{P} = \langle P, T', \mathcal{E}, m_0, \ell, F \rangle$, where $T' = T \cup \{\mathsf{zr}\}$, ℓ is an arbitrary injective labelling, and F is the set of all markings (P, $\mathcal{E}$, and m are as above). Intuitively, $\mathcal{P}$ is just G extended with a "do nothing" transition zr, which can always fire.

We define a counting automaton $A = \langle \{q\}, n, q, \mathbf{k}_0, \delta, \Omega \rangle$ from G as follows. Define $n \triangleq |P|$, and let $\mathit{place} : \{1, ..., n\} \to P$ be a bijection. The initial vector is defined to be $\mathbf{k}_0 \triangleq (k_1, ..., k_n)$, where for each i,

$$k_i = m_0(\mathit{place(i)})$$

The transition function is defined, for any $c \in \Sigma$ such that there is some $t \in T$ with $\ell(t) = c$, by

$$\delta(q, c) \triangleq \langle q, (\alpha_1, ..., \alpha_i) \rangle$$

where for each i

$$\alpha_i = \begin{cases} \mathsf{tst} & \text{if } \mathit{place}(i) \in {}^{\bullet}t \cap t^{\bullet} \\ \mathsf{dec} & \text{if } \mathit{place}(i) \in {}^{\bullet}t \\ \mathsf{inc} & \text{if } \mathit{place}(i) \in t^{\bullet} \\ \mathsf{nop} & \text{otherwise} \end{cases}$$

We also add another transition to δ for the distinguished "do nothing" transition zr, except that instead of doing nothing, we check that p is non-zero: with one exception: we define

$$\delta(q, \ell(\mathsf{zr})) \triangleq \langle q, (\alpha_1, ..., \alpha_n) \rangle$$

$$\alpha_i = \begin{cases} \mathsf{tst} & \text{if } \mathit{place}(i) = p \\ \mathsf{nop} & \text{otherwise} \end{cases}$$

Last, we define Ω by

$$\Omega(q) \triangleq \{(k_1, ..., k_n) \in \mathbb{N}^n : \exists m \in F. \forall i. k_i = m(\mathit{place}(i))\}.$$

We define an annotation φ for A that is trivially inductive by taking $\varphi(q) \triangleq \mathit{true}$. We have that $\langle \varphi, A \rangle$ is a counting proof.

It is easy to see that the control flow net $\mathcal{P}$ and the counting automaton A act exactly the same, except at markings where $\mathcal{P}$ can fire zr, but A cannot. Such a marking is reachable from m (in $\mathcal{P}$ and A) iff p is zero reachable m (in G), thus completing the proof. □

While the decision problem for proof checking has high computational complexity, from the standpoint of certifying the correctness of a program we may be satisfied with a faster semi-decision procedure. For example, acceleration has proved to be an effective technique in practice for Petri net reachability [7]. We also note a recent result by Leroux [35]: if a given marking of a Petri net is not reachable, there exists a Presburger-definable inductive invariant for the Petri net. This allows for the possibility of using well-known reachability algorithms for (sequential) integer programs (e.g., [12, 39]) to be used for counting proof checking.

7. Completeness

We now show the (relative) completeness of our counting proof method, when programs are modelled as control flow nets. First, we observe that proving that a control flow net $\mathcal{P}$ satisfies a given specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$ is equivalent to proving that a particular *flattened* program $\mathcal{P}^\flat$ satisfies $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$. The flattened program is a sequential program with extra natural-typed variables which are used to represent a marking. We then show that any inductive assertion for $\mathcal{P}^\flat$ can be (trivially) transformed into a counting proof. The relative completeness of counting proofs thus follows from the relative completeness of the inductive assertion method. We now formalize this argument.

Let $\mathcal{P} = \langle P, T, \mathcal{E}, \ell, m_0, F \rangle$ be a control flow net. A *configuration* of $\mathcal{P}$ is a pair $\langle m, s \rangle$, where m is a marking and $s \in S$ is a memory state. The *flattened program* $\mathcal{P}^\flat$ is a transition system where the state space is the set of configurations of $\mathcal{P}$ and the transition relation is

$$\langle m, s \rangle \to \langle m', s' \rangle \iff \exists t \in T. m \xrightarrow{t} m' \text{ and } \langle s, s' \rangle \in [\![\ell(t)]\!]$$

A *flattened assertion* is a formula in Φ^P (assertions over an extended vocabulary that includes an additional variable symbol p of sort $\mathbb{N}$ for each place $p \in P$). A configuration $\langle m, s \rangle$ of $\mathcal{P}$ can be viewed as a structure for this vocabulary (in the model-theoretic sense) by interpreting each new variable symbol p as $m(p)$.

For flattened assertions ψ, ψ' and a transition $t \in T$, we write

$$\{\psi\}\ t\ \{\psi'\}$$

if for all configurations $\langle m, s \rangle$ and $\langle m', s' \rangle$ such that $\langle m, s \rangle \models \psi$ and $\langle m, s \rangle \xrightarrow{t} \langle m', s' \rangle$, we have $\langle m', s' \rangle \models \psi'$.

Definition 7.1 Let $\mathcal{P} = \langle P, T, \mathcal{E}, \ell, m_0, F \rangle$ be a control flow net and let $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$ be a specification. A *global inductive assertion* is a flattened assertion ψ for which the following hold:

- $\psi_{\mathsf{pre}} \wedge \varphi_{m_0} \models \psi$
- $\psi \wedge \varphi_F \models \psi_{\mathsf{post}}$
- For all $t \in T$, $\{\psi\}\ t\ \{\psi\}$

where φ_{m_0} is a formula defining the initial marking m_0 and φ_F is a formula defining the set of final markings F. $\lrcorner$

Theorem 7.2 Let $\mathcal{P} = \langle P, T, \mathcal{E}, \ell, m_0, F \rangle$ be a control flow net and let $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$ be a specification. If there exists a global inductive proof that $\mathcal{P}$ satisfies the specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$, then there exists a counting proof. $\lrcorner$

Proof. Let $\mathcal{P} = \langle P, T, \mathcal{E}, \ell, m_0, F \rangle$ be a control flow net and let $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$ be a specification. Suppose that ψ is a global inductive assertion for $\mathcal{P}^\flat$ that proves that $\mathcal{P}$ satisfies the specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$.

We construct a counting automaton A that is equivalent to $\mathcal{P}$ as in the proof of Theorem 6.2. Consider the formula ψ' obtained by replacing each place variable p with its associated counter variable $\mathtt{k}_i$. It is easy to check that $\langle A, \lambda q.\psi' \rangle$ is a counting proof that proves that $\mathcal{P}$ satisfies the specification $\psi_{\mathsf{pre}}/\psi_{\mathsf{post}}$. □

8. Discussion Through Examples

In this section, we use examples to have a more in-depth discussion about a few points about the counting proofs framework. These points include:

- Control flow nets are a very general mechanism for representing programs with infinite control, no matter what the source of infinity of the control may be (Section 8.1).
- The structure of counting proof is in some sense independent of the control flow structure of the original program (Section 8.1).
- Control flow nets encode a verification problem. That is, they represent an integration of the control flow of the program and the correctness property. An adequate representation of program and property by a control flow net may not exist, and if it does, it may be difficult to build (Section 8.2).

8.1 Tree Traversal Example

```
Traverse(node) {
   if (node == null)
      return;
   visit(node);
   Traverse(node.left);
   Traverse(node.right);
}
```

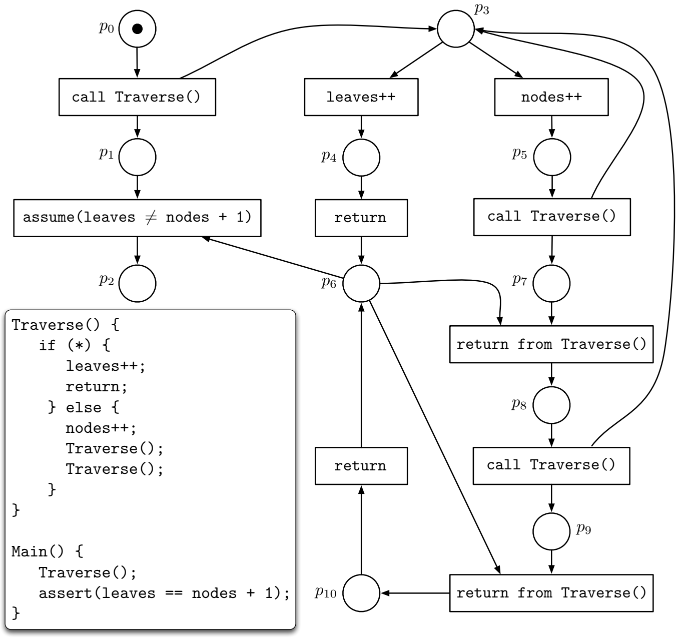
On the right is a recursive (preorder) tree traversal routine. We will give two implementations of this program, one a sequential recursive program and the other a parallel recursive program, to illustrate two important features of the counting proofs framework: first, control flow nets are a very general mechanism for representing programs with infinite control; and second, counting proofs are in some sense independent of the control flow structure of the original program.

We will use two different implementations of the generic traversal template from above: (i) a sequential recursive implementation, and (ii) a parallel and recursive implementation, to demonstrate the generality of the control flow nets, and the independence of counting proofs' structure from that of the program. We diverge slightly from the presentation of counting proofs and control flow nets from the preceding sections by using an acceptance condition based on linear arithmetic formulae rather than upwards closed sets (i.e., the set of accepting vectors of a counting proof and the set of accepting markings of a control flow net can be described by a QF_LRA formula rather than just an upwards closed set). As noted in Section 6, proof checking is decidable for this class (and in fact, the more general class of semi-linear sets). It is easy to see that the proof synthesis procedure from Section 4 can be adapted to this more general setting as well.

Recursive Traverse

We first look at a variation of the code above which is a sequential recursive program to demonstrate how recursive programs (which have non-regular trace languages) can be modelled using control flow nets. It is known that Petri net languages are incomparable with context free languages. However, they do properly include (and the inclusion is proper) the set of bounded context free languages, which is the most general class for which decidability results have been proved [21].

Figure 4 includes a simplified implementation of `Traverse` which abstracts away the heap manipulation operations, but keeps the relevant control information. We introduce two global counters to count the number of `leaves` and internal `nodes` that are visited by the `Traverse()` routine. If `nodes` and `leaves` are initially set to 0, then after `Traverse` is finished, for any binary tree the property `leaves = nodes + 1` is true.



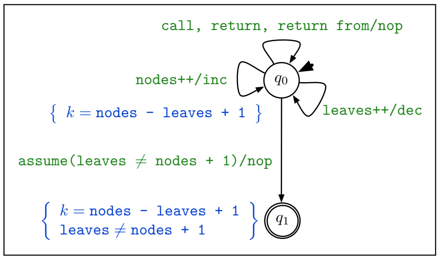
```
Traverse() {
   if (*) {
      leaves++;
      return;
   } else {
      nodes++;
      Traverse();
      Traverse();
   }
}

Main() {
   Traverse();
   assert(leaves == nodes + 1);
}
```

Figure 4. Sequential `Traverse` code and control flow net. The control flow net accepts when a token is at p_2 and there are no other tokens.

Figure 4 also illustrates the control flow net for this program. A call statement is modelled by two transitions: a `call` transition and a `return from call` transition. The call adds a token at the place p_3 (the entry location of `Traverse`), to trigger another execution of the method body. However, the current execution is blocked from progressing (note the incoming edge from place p_6 to the `return from call` statement) until a return token is provided through p_6. The erroneous traces (and therefore those in the language of the control flow net) are those that put a token at place p_2 (i.e. when the assertion is violated), and all other places contain zero tokens. The latter ensures that the all executions of all pending frames of `Traverse` are completed before the property is checked to hold. The control flow model allows several partial executions of different frames to co-exist, and the property does not hold until they all finish.

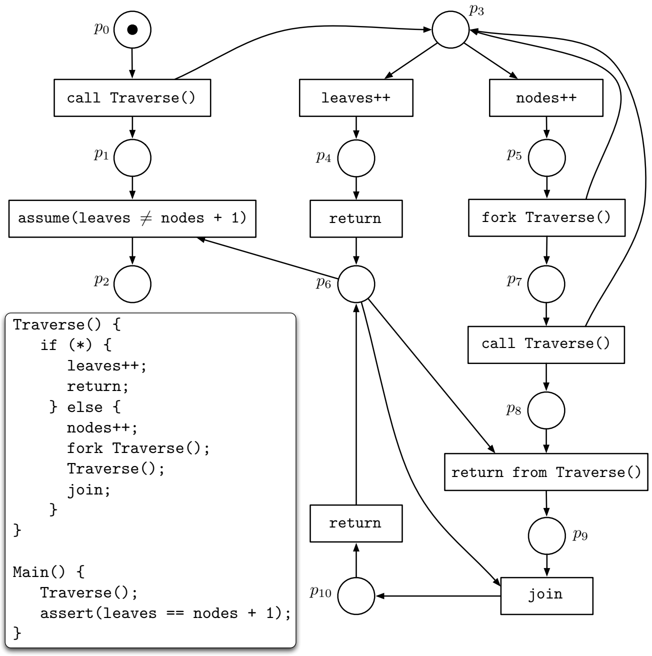
Proof of Correctness for Traverse

The proof of the correctness of the recursive `Traverse` appears on the right. It is a counting automaton with an additional counter variable k, which counts the difference between `nodes+1` and `leaves`. Note that the initial value of this counter is 1 (rather than 0, as we have seen in previous examples). When `Traverse` returns to the `Main` procedure and we check the assertion condition `count` and proceed to error location (i.e., we execute `assume(leaves != nodes + 1)`), we move to the state q_1 and stop reading additional statements. The automaton



call, return, return from/nop
nodes++/inc
q_0
leaves++/dec
$\{ k = \mathtt{nodes - leaves + 1} \}$
assume(leaves ≠ nodes + 1)/nop
$\left\{ \begin{array}{l} k = \mathtt{nodes - leaves + 1} \\ \mathtt{leaves} \neq \mathtt{nodes + 1} \end{array} \right\}$
q_1

accepts when the final value of the counter k is 0, which implies that $0 = k = \mathit{nodes} - \mathit{leaves} + 1$, contradicting the assumption $\mathit{leaves} \neq \mathit{nodes} + 1$.



```
Traverse() {
   if (*) {
      leaves++;
      return;
   } else {
      nodes++;
      fork Traverse();
      Traverse();
      join;
   }
}

Main() {
   Traverse();
   assert(leaves == nodes + 1);
}
```

Figure 5. Parallel `Traverse` code and control flow net. The control flow net accepts when a token is at p_2 and there are no other tokens.

Recursive and Parallel Traverse

Figure 5 presents a variation of the traverse implementation from Figure 4 in which the two recursive calls to `Traverse` are executed in parallel by forking a new thread for one of them. This program is both concurrent and recursive with both unbounded recursion and unbounded parallelism, which puts it in a class of very difficult programs to verify.

The control flow net for the parallel version is also depicted in Figure 5. Note that the join statement expects a token provided after the completion of the execution of the forked thread to continue, as with `return from call` in the recursive case.

A remarkable fact is that the proof of correctness of this version of `Traverse` is still the same proof that was presented in the previous section for the recursive version (with the minor addition of the `join` action to the set of actions `call, return, return from` that are labelling the self loop on q_0). Despite the fact that the two programs have substantially different control flow structure, the counting arguments for the correctness of their set of traces are the same. This is due to the fact that the proof is constructed based on a set of program traces, and independent of the control structure. Basically, in both cases the same counting argument applies for the correctness of program traces. It is a rather interesting feature to be able to *reuse* a proof of correctness when a program is changed, for example, for performance reasons (as is the case when we parallelize `Traverse`).

8.2 Ticket Algorithm

Our approach assumes that the alphabet of program statements is finite. If an unbounded number of threads execute a program over local variables, we need an infinite alphabet of statements (each program statement that manipulates a local variable, must be copies for each thread). However, our approach may still be applicable by using a *symmetry* argument. This section gives an example of such an argument.

We consider a well-known mutual exclusion protocol for an unbounded number of processes, namely the *ticket algorithm.* At each instant, a ticket counter t contains the value for the next available ticket, and a service counter s contains the number of already serviced clients. A client can acquire a ticket by setting its own local ticket number m to the current value of the ticket counter t, after which t is incremented by 1 to reflect the available ticket for the next client (`t++`). When a local ticket number m is equal to or smaller than the service number s, the client can enter the *critical* section (`assume(m <= s)`). When this customer exits the critical section, the service number s is incremented by 1 (`s++`).

The `Ticket` program consists of an arbitrary number of threads which each execute the same code, pictured to the right. The variables s and t are global (initially, $s = t = 0$). The variable m is local to each thread. The mutual exclusion property is that at most one thread can be at `crit`, or equivalently, while some thread T_1 is still at `crit`, another thread T_2 cannot enter `crit`.

```
idle:      m = t++
wait:      assume(m <= s)
crit:      s++
```

Interestingly, it does not seem possible to encode the verification problem for mutual exclusion by a control flow net (even after using a symmetry argument). Instead of mutual exclusion, we will consider a stronger correctness property (i.e., a property that implies mutual exclusion): if a thread T_2 has requested a ticket after another thread T_1, then T_2 cannot enter `crit` while T_1 is still at `wait` or `crit`. For brevity we refer to this property as FCFS, First Come First Serve.

Our encoding of the FCFS property relies on the notion of *minimal error traces*. Consider an error trace τ which violates FCFS. We call τ *minimal* if no proper prefix of τ already violates FCFS. To prove correctness, it is sufficient to show that every minimal error trace is infeasible.

We can decompose every minimal error trace as follows (we mark a statement by the thread that executes it):

$$\tau; [T_1\colon \texttt{m}_2\texttt{=t++}]; \tau'; [T_2\colon \texttt{m}_1\texttt{=t++}]; \tau''; [T_2\colon \texttt{assume(m}_1\texttt{<=s)}]$$

where $[T_1\colon \texttt{s++}]$ does not occur in τ, in τ', or in τ''. That is, every minimal error trace contains, in order, the request of T_1, the request of T_2, and, at the last position, the enter of T_2, and thread T_1 may or may not enter after its request, but if it does, it cannot not exit.

In the setting above, the thread T_2 is the *bad thread.* We now use a symmetry argument: without loss of generality, we may assume that the bad thread is thread 0. The reason that this is no loss of generality, is due to the symmetry of `Ticket`. For any minimal error trace where the bad thread is some thread other than thread 0, we may simply swap that thread and thread 0 and arrive at another minimal error trace, one where the bad thread is thread 0. Thanks to this symmetry, it is sufficient to show that all such 0-distinguished minimal error traces are infeasible.

We finitize the alphabet of program statements by "forgetting" the local variables of every *environment thread* (i.e., a thread other than thread 0). The statement `m = t++` of an environment thread becomes `_ = t++` (which is semantically the same as `t++`), and likewise `assume(m <= s)` becomes `assume(_ <= s)` (which is semantically the same as `assume(true)`). It is sufficient to show that every 0-distinguished minimal error trace over the finite alphabet of program statements is infeasible (since forgetting local variables can only make more traces feasible).

The set of traces of the control flow net in Figure 6(a) contains all 0-distinguished minimal error traces over the finite alphabet of program statements (incidentally, it contains also some non-minimal ones). The transitions and places are arranged in three columns. The left most column corresponds to environment threads that acquire their ticket before thread 0, the second column corresponds to the distinguished thread 0, and the last column corresponds to environment threads that acquire their ticket after thread 0.

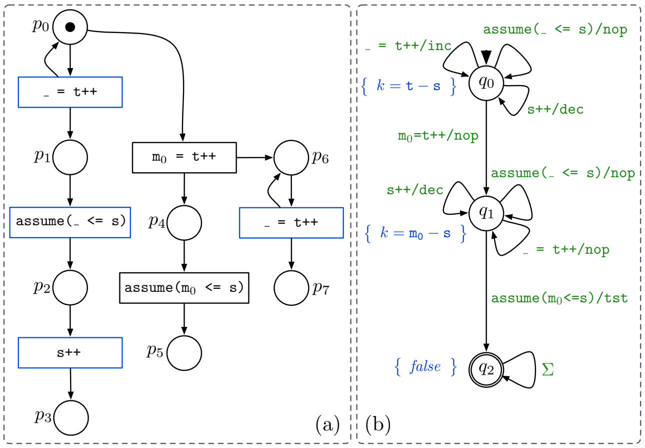


Figure 6. `Ticket`: (a) Control flow net. The set of final markings is defined by $p_1 + p_2 \geq 1 \wedge p_5 \geq 1$. (b) Counting proof.

Counting Proof of Ticket The counting automaton shown in Figure 6 has three states. The initial state is q_0, its only final state is q_2. The initial value of the counter k is 0. While in state q_0, the automaton counts the request statements (positively) and the exit statements of environment threads (negatively); it does not count enter statements. While in state q_1, it counts the exit statements of environment threads (negatively); it counts neither request statements nor enter statements. The request statement of the distinguished thread leads from q_0 to q_1. The enter statement of the distinguished thread leads from q_1 to q_2 if the `tst` operation succeeds, i.e., if $k \geq 1$.

$$\boxed{\begin{aligned} p_1 + p_2 &\leq k \\ p_0 &\leq q_0 \\ p_4 &\leq q_1 \\ p_5 &\leq q_2 \vee k \leq 0 \end{aligned}}$$

Proof Checking We use this example to convey our intuition that proving the inclusion of the language of a control flow net in the language of a counting automaton often involves a set of very simple invariants. The formula pictured to the right is a linear arithmetic formula which relates configurations of the control flow net with configurations of the counting proof (two configurations are related if they satisfy the formula to the right; the control flow net configuration is used to interpret p variables and the counting automaton configuration is used to interpret k and the q variables; the value of q is 1 or 0). One may check that this relation is a kind of simulation: whenever the control flow net may take a step, the counting automaton may take a corresponding step *or* the control flow net is in a "dead" configuration that cannot reach a final marking. Moreover, the initial and final configurations of the control flow net and counting automaton are related. Thus, this formula shows that the traces of the control flow net are accepted by the counting automaton.

9. Related Work

Extensive research has been done about verification of parameterized systems. Apt and Kozen [4] argued that with little capabilities granted to individual processes, the verification problem becomes undecidable. There has since been a great deal of study of decidability/undecidability of subclasses of parameterized systems [32, 38]. There is a good survey [49] that covers a lot of the existing techniques for verification of parameterized systems. We emphasize here that our goal is not to verify parameterized distributed protocols, but programs; and therefore, we do not mention a lot of related work that is focused on the intricacies of verifying these protocols.

Deductive techniques We are aware of two proof systems that are applicable to parameterized systems: (1) a parameterized Owicki-Gries type proof system [40], in which if the assertions (and the auxiliary variables) are provided by the user as annotations, then the proof checking is done mostly automatically, and (2) The QED [17] system that is based on the Lipton's reduction by inferring atomic blocks, which works independently of how many threads are running in parallel. Both of these systems need (partial or total) user-provided annotations to prove programs correct.

Induction-based techniques There are techniques based on induction (on the number of processes) [33, 36], that rely on finding an abstraction and approximation of *network invariants* [13, 14, 31].

The method of *invisible invariants* [5, 44] automatically generates inductive assertions for the verification of safety properties of parameterized systems. First, the symmetry of parameterized systems is exploited to *guess* a universally quantified assertion that over-approximate the set of reachable states. then, a *small model theorem* establishes when the assertions over parameterized systems can be model checked on small instantiations (whose size depend on the system and the assertions) to derive the validity over any instantiation. This method is limited, however, to systems with finite data, and the application of it in some contexts is far from trivial.

Regular model checking The idea, which was introduced in [48], is to represent the set of reachable states of a parameterized system using a regular language. Processes are assumed to be finite-state to give rise to a finite alphabet (alphabet letters correspond to processes' states). Transducers are used to approximate (through acceleration) the transitive closure of transition relation, and hence compute all reachable configurations of the systems. A lot of research has been done in improving the laborious problem of computing the meta transitions [1, 11, 36, 43] (including techniques based on acceleration, transitive closure, and widening), and even going beyond regular languages in [20]. The restriction that processes need to be finite state persists in all these extensions. The focus of this line of research has been on the verification of protocols, and hence, they often have much richer input languages (than the one we use in this paper) that are beyond our scope of parameterized programs; e.g. the use of existentially or universally quantified guards, which is not normally found in software systems.

Abstraction-based techniques In [27], a system of N processes communicating through finite-domain shared variables is abstracted by mapping the state systems into tuples where each dimension (having the values, 0, 1, or many) corresponds to the number of processes at a given local state, and additional dimensions are used to capture the value of global variables (finitely many possibilities). The resulting finite-state system was then verified by model checking (finite-state verification). In this and other *counter abstraction* techniques, a concrete state is counter abstracted by counting the number of processes in each local state [37, 45] limiting the counter to at most 2. Counter abstraction is simple to apply and when applicable works well. Its shortcoming is that it is only applicable to systems where each process has a small number of individual local states. More recently, in [28] symbolic techniques are used to overcome this problem and allow verification processes with larger number of local states. A new form of counter abstraction is discussed in [28] using parametric interval abstraction that facilitates verification of fault-tolerant distributed algorithms.

There is a class of techniques that are not strictly counter abstraction in the above sense (since the resulting abstract system is not finite), but have a similar flavour since they use the same counters, but untruncated. In [22], German and Sistla consider a parameterized system of processes that communicate synchronously, and show how to verify single-index properties. They achieve this by encoding the problem as a Petri net safety property and using Karp-Miller's coverability tree construction [30]. Similar abstraction-based techniques have been applied to verification of multi-threaded C [6], and Java [16] programs. In all these instances, processes are abstracted into finite-state processes, and the only source of infinity remains the existence of unboundedly many of these finite-state processes.

In [2], a CEGAR algorithm for parameterized systems is presented. The program model in this paper is very similar to ours (although the technique proposed in [2] is focused on proving safety for Petri net programs where the data variables are integers). The reachability algorithm employed by [2] is a fairly standard backward coverability algorithm - the insight of this paper is that the well quasi-order used for this algorithm can be refined using counter examples. The authors do not approach the problem of automatically synthesizing auxiliary variables.

In [8], it is shown how to represent a parameterized system of finite-state processes in the decidable logic WS1S, i.e. the current state of the system is modelled as a fixed number of finite subsets of natural numbers and the transitions of processes are described in WS1S formulae. Later, in [9], they extend the method, using a combination of theorem proving and the algorithmic techniques from [8], so that each process can have an unbounded state space. The theorem proving side helps prove a simulation relation between the original system and a so-called doubly-parametric system which is restricted enough to be expressible and checkable within the original framework from [8].

The problem of proving data structure invariants for programs with unboundedly many threads is attacked in [10] and [47]. [10] 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 [47], in which a universally quantified *environment assertion* is used to keep track of relationships between a distinguished thread and all other threads. [18] is another technique that, like [10, 47], loses variable correlations in the interests of speed. Unlike [10] and [47], [18] is designed to compute numerical invariants rather than data structure invariants.

The conditions that need to be satisfied by the abstraction are restrictive enough that, as an example, the ticket example discussed here cannot be handled by the method.

The technique of trace abstraction has previously been discussed in [25] and in [19]. There, the sets of traces of sequential programs respectively concurrent (non-parameterized) programs were abstracted by regular languages. The issue of auxiliary variable synthesis was not investigated in that work, although it is potentially interesting also for sequential and concurrent (non-parameterized) programs.

Cut-off detection There is a strong belief (backed by empirical evidence) that parameterized systems often enjoy a small model property. This belief has given rise to a collection of techniques for verification of parameterized concurrent systems [3, 29, 34]. More precisely, analyzing a small number of processes (the so-called *cut-off* points) and their interactions is sufficient to determine the reachability of any bad states. In [29, 34], parameterized Boolean programs (more specifically the result of predicate abstraction of device drivers) are analyzed. The technique in [29] is complete with respect to Boolean programs (note that this does not carry over to the original infinite-state C programs), while the one in [34] is even incomplete for Boolean programs. More recently, in [3], communication protocols are the focus, and an abstraction scheme is used that attempts to detect the cut-off points dynamically during the verification procedure to stop the search. Here, also, the processes are assumed to be finite state.

10. Conclusion

In this paper, we introduced *counting proofs*, a new system for proving safety properties for programs with infinite control. We believe that counting proofs have independent interest, outside of the software verification algorithm presented here. For example, we believe it may have applications to analysis of black-box systems, where the proof checking problem cannot be done, but we may be able to make other guarantees (e.g., testing coverage). Another potential direction for future work is application is white-box verification where the proof checking problem is undecidable in general, but an incomplete semi-test can be used for proof checking.

This paper shows that a particular class of auxiliary variables – counters – can be synthesized automatically. A natural question is to ask what other classes of auxiliary variables admit synthesis procedures. Another question is whether basic ideas introduced in this paper can be applied to auxiliary variable synthesis in other proof systems, such as Owicki-Gries [41]. The problem of auxiliary variable synthesis is largely unexplored, and this paper takes a step in the direction.

References

[1] P. A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parameterized system verification. In *CAV*, pages 134–145, 1999.

[2] P. A. Abdulla, Y.-F. Chen, G. Delzanno, F. Haziza, C.-D. Hong, and A. Rezine. Constrained monotonic abstraction: a cegar for parameterized verification. In *CONCUR*, pages 86–101, 2010.

[3] P. A. Abdulla, F. Haziza, and L. Holík. All for the price of few. In *VMCAI*, pages 476–495, 2013.

[4] K. R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[5] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. D. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, pages 221–234, 2001.

[6] T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS*, pages 158–173, 2001.

[7] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast: acceleration from theory to practice. *Int. J. Softw. Tools Technol. Transf.*, 10(5): 401–424, Sept. 2008.

[8] K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S systems to verify parameterized networks. In *TACAS*, pages 188–203, 2000.

[9] K. Baukus, K. Stahl, S. Bensalem, and Y. Lakhnech. Networks of processes with parameterized state space. *Electr. Notes Theor. Comput. Sci.*, 50(4):386–400, 2001.

[10] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, pages 399–413, 2008.

[11] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, pages 403–418, 2000.

[12] A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.

[13] E. M. Clarke, O. Grumberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. In *PODC*, pages 240–248, 1986.

[14] E. M. Clarke, O. Grumberg, and S. Jha. Veryfying parameterized networks using abstraction and regular languages. In *CONCUR*, pages 395–407, 1995.

[15] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification*, pages 420–432, 2003.

[16] G. Delzanno, J.-F. Raskin, and L. V. Begin. Towards the automated verification of multithreaded java programs. In *TACAS*, pages 173–187, 2002.

[17] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15, 2009.

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

[19] A. Farzan, Z. Kincaid, and A. Podelski. Inductive data flow graphs. In *POPL*, pages 129–142, 2013.

[20] D. Fisman and A. Pnueli. Beyond regular model checking. In *FSTTCS*, pages 156–170, 2001.

[21] P. Ganty, R. Majumdar, and B. Monmege. Bounded underapproximations. *Formal Methods in System Design*, 40(2):206–231, 2012.

[22] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.

[23] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, pages 331–344, 2011.

[24] M. Hack. *Decidability questions for Petri nets*. PhD thesis, MIT, June 1976.

[25] M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of trace abstraction. In *SAS*, pages 69–85, 2009.

[26] L. E. Holloway, B. H. Krogh, and A. Giua. A survey of petri net methods for controlled discrete eventsystems. *Discrete Event Dynamic Systems*, 7(2):151–190, Apr. 1997.

[27] C. N. Ip and D. L. Dill. Verifying systems with replicated components in murφ. *Formal Methods in System Design*, 14(3):273–310, 1999.

[28] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *FMCAD*, pages 201–209, 2013.

[29] A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, pages 645–659, 2010.

[30] R. M. Karp and R. E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969.

[31] Y. Kesten, A. Pnueli, E. Shahar, and L. D. Zuck. Network invariants in action. In *CONCUR*, pages 101–115, 2002.

[32] S. R. Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *STOC*, pages 267–281, 1982.

[33] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. *Inf. Comput.*, 117(1):1–11, 1995.

[34] S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV*, pages 629–644, 2010.

[35] J. Leroux. Vector addition system reachability problem: a short self-contained proof. In *POPL*, pages 307–316, 2011.

[36] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *POPL*, pages 346–357, 1997.

[37] B. D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs i. *Acta Inf.*, 21:125–169, 1984.

[38] M. Maidl. A unifying model checking approach for safety properties of parameterized systems. In *CAV*, pages 311–323, 2001.

[39] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.

[40] L. P. Nieto. Completeness of the Owicki-Gries system for parameterized parallel programs. In *IPDPS*, page 150, 2001.

[41] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19:279–285, May 1976. ISSN 0001-0782.

[42] E. Pelz. Closure properties of deterministic petri nets. In *STACS*, pages 371–382, 1987.

[43] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *CAV*, pages 328–343, 2000.

[44] A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, pages 82–97, 2001.

[45] A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, ∞)-counter abstraction. In *CAV*, pages 107–122, 2002.

[46] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.

[47] M. Segalov, T. Lev-Ami, R. Manevich, R. Ganesan, and M. Sagiv. Abstract transformers for thread correlation analysis. In *APLAS*, pages 30–46, 2009.

[48] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *CAV*, pages 88–97, 1998.

[49] L. Zuck and A. Pnueli. Model checking and abstraction to the aid of parameterized systems (a survey). *Comput. Lang. Syst. Struct.*, 30 (3-4):139–169, Oct. 2004.