Verification of Concurrent Programs Using Petri Net Unfoldings

Daniel Dietsch, Matthias Heizmann, Dominik Klumpp(✉), Mehdi Naouar, Andreas Podelski, and Claus Schätzle

University of Freiburg, Freiburg im Breisgau, Germany
klumpp@informatik.uni-freiburg.de

Abstract. Given a verification problem for a concurrent program (with a fixed number of threads) over infinite data domains, we can construct a model checking problem for an abstraction of the concurrent program through a Petri net (a problem which can be solved using McMillan's unfoldings technique). We present a method of abstraction refinement which translates Floyd/Hoare-style proofs for sample traces into additional synchronization constraints for the Petri net.

Keywords: Petri nets · Unfoldings · Concurrency · Verification

1 Introduction

The verification of concurrent programs is an active topic of research, and since it is also an old topic of research, there is a large body of literature covering a wide area of aspects of the problem; see, e.g., [2,3,7,8,10,11,15,17,19,21]. In this paper, we address the verification problem for programs composed of a fixed number of threads over an infinite data domain.

This verification problem poses two major challenges. First, the challenge of interleavings. In contrast to sequential programs, the control-flow of concurrent programs is much more permissive: an execution trace is not a cohesive sequence of statements from one process but an interleaved sequence of execution traces of all processes. Hence we have to account for a gigantic number of possible orderings of statements of the system's independent processes. For finite state systems the problem has been successfully approached by *Petri net unfoldings*. If the finite state system is represented by a bounded Petri net (i.e., a Petri net where each place can only take a pre-defined fixed amount of tokens), an unfolding is a mathematical structure that allows us to see all reachable states without exploring all interleavings explicitly. Unfoldings explicitly preserve the concurrent nature of Petri nets and can be exponentially more concise than a naive reachability graph.

The second challenge that we are facing is that our variables take their value from an infinite data domain and hence we cannot directly apply algorithms for finite state systems. For sequential programs this second challenge is often approached by abstracting the program by a finite state system. If such

© Springer Nature Switzerland AG 2021

an abstraction represents all executions of the program but does not have any erroneous execution, we know that the program is correct. Finding a suitable abstraction is difficult. Algorithms for finding abstractions usually follow the counterexample-guided abstraction refinement scheme (CEGAR). There, the algorithm constructs abstractions iteratively. In each iteration the algorithm uses Floyd/Hoare-style annotations obtained from counterexamples to refine the abstraction.

In this paper, we present a method of abstraction refinement which, given a verification problem for a program composed of a fixed number of threads over an infinite data domain, constructs a model checking problem for a Petri net. The idea is to translate Floyd/Hoare-style annotations into synchronization constraints; by adding synchronization constraints to the Petri net, we refine the abstraction of the concurrent program through the Petri net. In summary, the method of abstraction refinement constructs a bounded Petri net and thus gives us the possibility to use Petri net unfoldings for the verification of programs composed of a fixed number of threads over an infinite data domain.

Let us motivate our approach by illustrating shortcomings of naive sequentialization, a straightforward approach to the verification of concurrent programs. Sequentialization means that we translate the concurrent program into a sequential program which allows us to apply all verification techniques for sequential programs. In its most basic form, the sequentialization produces a control flow graph (CFG) that is the product of the CFGs of the concurrent program's threads. However, this basic approach does not scale well: The product CFG must explicitly represent the many different interleavings. Hence the number of locations in the CFG grows exponentially with the number of threads. As an example, consider the schema for a concurrent program shown in Fig. 1. Given a number N, this yields a concurrent program with N threads. After the variable `x` is initially set to 0, the different threads all repeatedly modify `x` for a nondeterministically chosen number of times. The control flow graph for each thread is simple, it only requires 3 locations (loop head, location between both assignments, and loop exit). But the resulting product CFG has 3^N locations, it grows exponentially in the number of threads. For large N, even the construction of the product CFG may thus run out of time and memory. In our approach we do not construct this product but a Petri net that has for each thread one token and whose places are the locations of all CFGs. Hence our Petri net grows only linearly in the number of threads.

This paper is organized as follows. In Sect. 2 we demonstrate our approach on the example above and another example. Section 3 introduces our notation and terminology for finite automata and Petri nets. We use Petri nets to introduce the considered verification problem formally in Sect. 4. In Sect. 5 we present our algorithm for this verification problem, and in Sect. 6 we present an automata-theoretic difference operation required by our verification algorithm. In Sect. 7 we discuss how the difference operation introduces synchronization constraints. Finally, we discuss related work in Sect. 8 and conclude with Sect. 9.

```
x := 0 // initialization
```

```
thread1() {
  while (*) {
    x := x + 1
    x := x * 1
  }
}
```

```
thread2() {
  while (*) {
    x := x + 2
    x := x * 2
  }
}
```

• • •

```
threadN() {
  while (*) {
    x := x + N
    x := x * N
  }
}
```

Specification: After all threads have terminated, the value of x is non-negative.

Fig. 1. A concurrent program schema with a scalable (but fixed) number of threads N

2 Examples

In this section we illustrate two aspects of our method of abstraction refinement. Our method takes as input a verification problem for a concurrent program (i.e., a program composed of a fixed number of threads over an infinite data domain). The program's control flow is represented by a bounded Petri net. The property to be verified is encoded as unreachability of a special *error place* ℓ_{err} in this Petri net. Our method proceeds by iteratively adding synchronization to the input Petri net, in order to represent *data constraints* over the infinite program state space, i.e., the constraints on the control flow that are due to updates and tests of data values.

We begin by examining the example in Fig. 1 a bit closer, and we will demonstrate on this example the strength of our approach: Through its lazy synchronization and the use of unfoldings, we verify the program efficiently, regardless of the number of threads. The second example will illustrate how our approach adds synchronization where necessary.

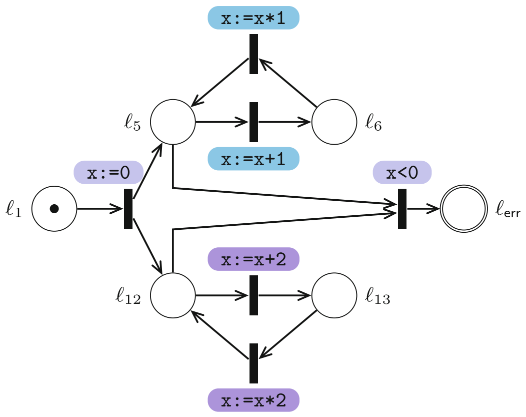
2.1 Retaining Concurrency of Different Threads

Consider again the concurrent program schema in Fig. 1. This program schema can be instantiated for any number of threads N. In Fig. 2a we see the instantiation of the schema in Fig. 1 for $N = 2$ threads. In our approach, we represent such a concurrent program in the form of a Petri net, in this case shown in Fig. 2b. Each transition of the Petri net is labeled with a statement of the concurrent program. Here, the first transition is labeled with the initialization statement for the global variable `x`. This transition starts the two threads. After some number of iterations, the threads can decide nondeterministically to exit their respective loops. Then, the last transition is enabled, which is labeled with the negated postcondition and leads to the *error place* ℓ_{err}. This Petri net is our initial abstraction of the concurrent program. The term *abstraction* refers to the fact that the actions labeling the transitions of the Petri net serve as reference; they are not interpreted for the operational semantics of the Petri net. The state of the Petri is purely defined by the number of tokens on each of its places. Hence, in Fig. 2b, the error place ℓ_{err} is reachable.

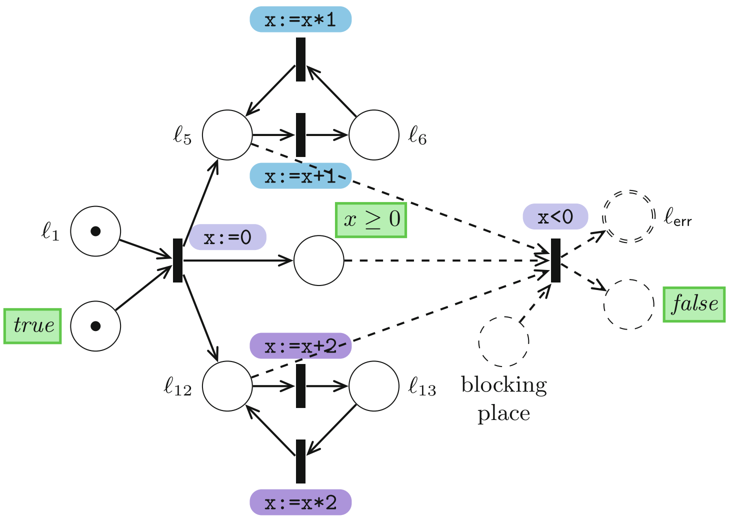
```
int x := 0;

thread1() {
  while (*) {
    x := x + 1
    x := x * 1
  }
}

thread2() {
  while (*) {
    x := x + 2
    x := x * 2
  }
}
```

Specification: When all threads have terminated, x is non-negative.

(a) Instance of the schema in Fig. 1 for $N = 2$ threads,



(b) A Petri net representing the verification problem posed by Fig. 2a. In this Petri net, an error state (a marking with a token on the place ℓ_{err}) is reachable.



(c) A refined version of the Petri net in Fig. 2b where synchronization has been added parsimoniously. The dashed arcs and places are unreachable.

Fig. 2. Example. Parsimoniously added synchronization (reflecting data constraints) reveals the unreachability of an error place in the program. Synchronization is avoided when the interleavings between actions of different threads are irrelevant in the abstract – even though they may be relevant in the concrete.

The verification task now consists of showing that, when the statement semantics *are* taken into account, no firing sequence that reaches ℓ_{err} actually corresponds to a program execution. We pick one such firing sequence and analyse the corresponding sequence of statements, taking into account the operational semantics of the statements. This yields the following two data constraints:

1. After executing the statement `x:=0`, the program is in a state where $x \geq 0$ holds.
2. If the program is in a state where $x \geq 0$ holds, it cannot execute the assume statement `x<0`.

Next, our algorithm constructs the Petri net depicted in Fig. 2c by adding synchronization that reflects the two data constraints above to the initial Petri net of Fig. 2b. The synchronization constraints are implemented by adding three additional places (labeled by *true*, $x \geq 0$ and *false*) that represent the knowlege about the program's data that we want to replicate. Intuitively, the places are used to abstract the program's data values. The transitions labeled `x:=0` and `x<0` are connected to the new places. The order in which the statements `x:=x+1`, `x:=x*1`, `x:=x+2` and `x:=x*2` are executed is relevant for the concurrent program (i.e., for the final value of `x`). It is, however, irrelevant for the correctness proof that uses the state assertion $x \geq 0$. Since these four statements are not relevant for establishing the state assertion $x \geq 0$, and this state assertion is preserved by these statements, our algorithm does not connect the transitions labeled with these statements with one of the new places.

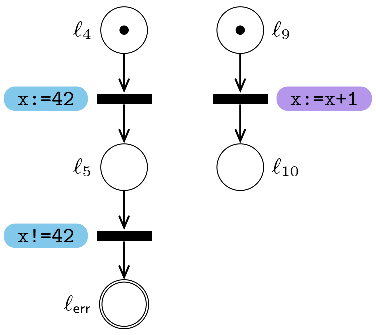
In the resulting *refined* Petri net Fig. 2c, the transition labeled `x:=0` can fire if there is a token in the *true* place, and moves this token to the place labeled $x \geq 0$. Now the transitions in the two threads can fire repeatedly, without moving the token in $x \geq 0$. When at some point both the places ℓ_5 and ℓ_{12} have a token, the transition labeled $x < 0$ could fire. However, this would put a token in the place labeled *false*, representing a violation of the data constraints. Hence we prevent this transition from ever firing by adding a *blocking place* as predecessor, which will never have a token. As a result, the place ℓ_{err} is unreachable, and we conclude that the concurrent program satisfies its specification.

Our approach proceeds in the same way for all instantiations of the concurrent program of Fig. 1, for every number of threads N: The state assertions *true*, $x \geq 0$ and *false* are added to the Petri net, and synchronized only with the statements `x:=0` and `x<0`. However, synchronization with each thread is not necessary. As a result, the size of the final refined Petri net grows only linearly in the number of threads N. Through the use of unfoldings, we can check the reachability of the error place efficiently, without explicitly considering all interleavings of transitions in the Petri net.

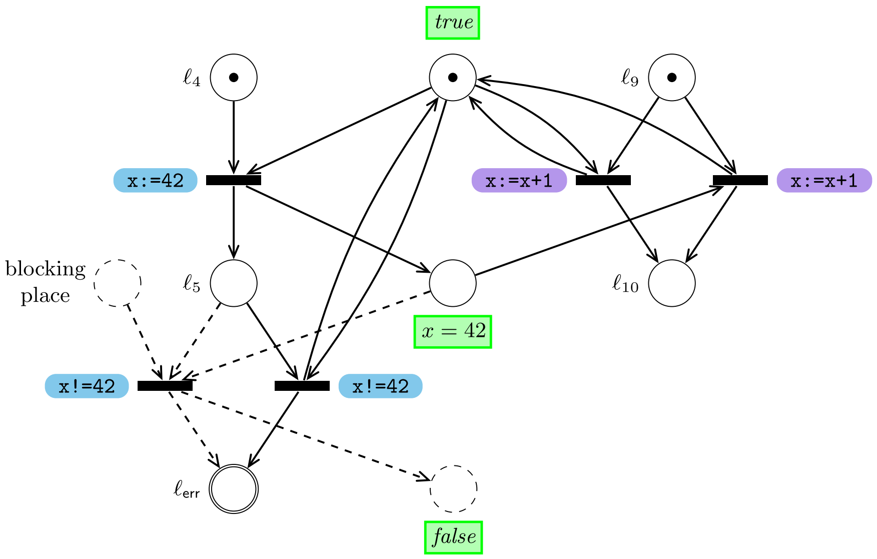
```
 1 int x;
 2
 3 thread1() {
 4   x := 42
 5   assert x == 42
 6 }
 7
 8 thread2() {
 9   x := x + 1
10 }
```

Specification: The statement `assert x == 42` never fails.

(a) Concurrent program whose threads operate on a shared variable `x`.



(b) Petri net representation of the program in Fig. 3a. In this Petri net, an error state (a marking with a token on the place ℓ_{err}) is reachable.



(c) Petri net constructed by adding synchronization constraints to the Petri net in Fig. 3b. Dashed lines indicate parts that are unreachable. The error place ℓ_{err} remains reachable.

Fig. 3. Example. Our approach represents the concurrent program in Fig. 3a as the Petri net in Fig. 3b. To this initial abstraction, we add synchronization reflecting *data constraints* on the control flow of the concurrent program. The resulting Petri net is shown in Fig. 3c.

2.2 Adding Synchronization Lazily

Consider now the concurrent program depicted in Fig. 3a. Here, the first thread sets a global variable `x` to the value 42, and then asserts that it holds said value. The second thread on the other hand increments `x`. We represent this program through the Petri net depicted in Fig. 3b. Once again, transitions are labeled by statements of the concurrent program, which only serve as reference; they are not interpreted for the operational semantics of the Petri net. The transition labeled with the negation of the assertion condition leads to the error place ℓ_{err}. The place ℓ_{err} is reachable by a firing sequence of the Petri net.

As before, our algorithm analyses the sequence of statements along a firing sequence that reaches ℓ_{err}, for instance the sequence `x:=42` `x!=42`. The analysis of this sequence of statements, taking into account the operational semantics of the statements, yields the following two data constraints:

1. After executing the assignment `x:=42`, the program is in a state where $x = 42$ holds.
2. If the program is in a state where $x = 42$ holds, the assertion condition is not violated, i.e., the assume statement `x!=42` cannot be executed.

Next, the algorithm constructs the Petri net depicted in Fig. 3c, by adding synchronization to the initial Petri net. As before, we add three additional places (labeled by *true*, $x = 42$ and *false*) to reflect our data constraints. However, in this example, the order and number of firings of the transitions in the two threads is not irrelevant to our data constraints: In particular, if the program is in a state where $x = 42$ holds and the second thread now executes `x:=x+1`, then it is no longer guaranteed that $x = 42$ holds. Hence, we must connect the transitions to the new places. We have two copies of the transition labeled `x:=x+1`: One copy can fire if there is a token in the *true* place, and puts the token back into the *true* place. The second copy can fire if there is a token in the place labeled $x = 42$, and moves that token into the *true* place. Similarly for the two copies of the transition labeled `x!=42`: One copy takes a token from the *true* place and puts it back, the other takes a token from the $x = 42$ place and moves it to the *false* place. This however would represent a violation of the data constraints, and thus we again add a *blocking place* to prevent this transition from firing. The transition labeled `x:=42` moves a token from the *true* place to the $x = 42$ place. We omit the second copy of this transition (with predecessor $x = 42$), as it would be unreachable.

In the resulting Petri net Fig. 3c, an execution of the Petri net labeled with the sequence `x:=42` `x!=42` is no longer possible. However, the ℓ_{err} place is still reachable through a firing sequence labeled with the statements `x:=42` `x:=x+1` `x!=42`. Hence our algorithm now analyses this sequence, taking into account the semantics, and determines that there are no data constraints preventing the execution of this sequence of statements. We conclude that the concurrent program is incorrect: The assertion may indeed be violated. Because of the introduced synchronization, the reachability check in this example has to explicitly consider the different orderings between transitions from the two threads. This explains our focus on avoiding synchronization wherever possible, in order to maintain the efficiency of our approach. However, in this case a separate consideration of the different orderings is inevitable, as exactly one of them can be executed and leads to an error.

3 Petri Net and Finite Automata

In this section we introduce our notation and terminology for Petri nets and finite automata. Analogously to finite automata we will introduce Petri nets as acceptors of languages. Throughout this paper we will only work with *bounded* Petri nets, but we will define a bounded Petri net as a special case of a (general) Petri net.

3.1 Finite Automata

A *finite automaton* $\mathcal{A} = (\Sigma, Q, \delta, q_{\mathsf{init}}, Q_{\mathsf{acc}})$ consists of an alphabet Σ, a finite set of states Q, a transition relation $\delta \subseteq Q \times \Sigma \times Q$, an initial state $q_0 \in Q$ and a set of accepting states $Q_{\mathsf{acc}} \subseteq Q$. The elements of Σ are called *letters*, and sequences $w \in \Sigma^*$ are *words*. We say that a word $w = a_1 \dots a_n \in \Sigma^*$ is accepted by $\mathcal{A}$ iff there exists a corresponding run of states $q_0 \dots q_n$ such that

- $q_0 = q_{\mathsf{init}}$ is the initial state,
- for all $i \in \{1, \dots, n\}$, it holds that $(q_{i-1}, a_i, q_i) \in \delta$,
- and $q_n \in Q_{\mathsf{acc}}$ is accepting.

The set of all words accepted by $\mathcal{A}$ is the language $L(\mathcal{A})$ recognized by the automaton.

We say that $\mathcal{A}$ is *deterministic* iff for all $q \in Q$ and $a \in \Sigma$, there exists at most one q' such that $(q, a, q') \in \delta$. Dually, we say that $\mathcal{A}$ is *total* iff there always exists at least one such q'. Hence, the transition relation of a deterministic total automaton is a function, and we write $\delta(q, a) = q'$ in place of $(q, a, q') \in \delta$. It is well-known that for every finite automaton $\mathcal{A}$, one can compute a deterministic total automaton $\mathcal{A}'$ that recognizes the same language, $L(\mathcal{A}) = L(\mathcal{A}')$. We abbreviate *deterministic total automaton* as *DFA*.

We call a transition $(q, a, q') \in \delta$ a *self-loop* iff $q = q'$.

3.2 Petri Nets as Language Acceptors

We define a Petri net as a 7-tuple $\mathcal{N} = (\Sigma, P, T, F, m_{\mathsf{init}}, \lambda, P_{\mathsf{fin}})$ where Σ is an alphabet, P are *places*, T are *transitions* with $P \cap T = \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*, $m_{\mathsf{init}} : P \to \mathbb{N}$ is an *initial marking*, $\lambda : T \to \Sigma$ is a *labeling* of transitions, and $P_{\mathsf{fin}} \subseteq P$ is a set of *accepting places*. We will sometimes use an infix notation for the flow relation and write e.g. $p \; F \; t$ instead of $(p, t) \in F$.

We define a *marking* as a map $m : P \to \mathbb{N}$ that assigns a token count to each place. We write M to denote that set of all markings over P. A marking $m \in M$ *covers* a place $p \in P$ iff m assigns at least one token to p.

$$m \text{ covers } p \Leftrightarrow m(p) > 0$$

We call a marking $m \in M$ *accepting* iff it covers at least one accepting place.

With $m \rhd_t m'$ we denote that transition $t \in T$ can be *fired* from marking m, i.e., all predecessor places have a token, and the firing of t results in the marking m'. Formally, we define the *firing relation* $\rhd \subseteq M \times T \times M$ as

$$m \rhd_t m' \;\Leftrightarrow\; \begin{array}{l} \forall p \in P : p\ F\ t \to m(p) > 0 \qquad \text{and} \\ \forall p \in P : m'(p) = m(p) - |\{t \in T \mid p\ F\ t\}| + |\{t \in T \mid t\ F\ p\}| \end{array}$$

A *firing sequence* in $\mathcal{N}$ is then an alternating sequence $m_0 \rhd_{t_1} m_1 \rhd_{t_2} \dots \rhd_{t_n} m_n$ of markings $m_i \in M$ and transitions $t_i \in T$, such that (a) $m_0 = m_{\mathsf{init}}$ is the initial marking and (b) the sequence adheres to the firing relation, i.e. $m_{i-1} \rhd_{t_i} m_i$ for all $i \in \{1, \dots, n\}$. A firing sequence ending in an accepting marking is called *accepting*. We say that a marking m is reachable iff there exists a firing sequence $m_0 \rhd_{t_1} m_1 \rhd_{t_2} \dots \rhd_{t_n} m_n$ with $m_n = m$.

We define the language that is recognized by a Petri net as follows:

$$L(\mathcal{N}) := \left\{ a_1 a_2 \dots a_n \in \Sigma^* \;\middle|\; \begin{array}{l} \exists \text{ accepting firing sequence} \\ \qquad m_0 \rhd_{t_1} m_1 \rhd_{t_2} \dots \rhd_{t_n} m_n \\ \text{such that } \forall\, i \in \{1, \dots n\} : \lambda(t_i) = a_i \end{array} \right\}$$

A net is *bounded* (also known as *1-safe* or just *safe*) iff all reachable markings have at most one token per place. In this paper we consider only bounded Petri nets and we will often use *Petri net* as a synonym for *bounded Petri net*. We identify markings $m : P \to \mathbb{N}$ with sets $m' \subseteq P$.

$$m \equiv m' \quad \Leftrightarrow \quad \forall p \in P : m(p) = 1 \leftrightarrow p \in m'$$

4 Petri Programs

In this section we describe our formal setting, based on the notion of Petri nets as language acceptors as presented in Sect. 3.2. We then make precise the verification problem solved by our algorithm.

4.1 Program Semantics

We assume a fixed set of program variables **Var** and a language of statements **Stmt**. A *program state* $s \in$ **State** maps program variables to their values, which may lie in an infinite data domain (such as $\mathbb{Z}$). Each statement $st \in$ **Stmt** is assigned a semantics $[\![st]\!] \subseteq$ **State** $\times$ **State**, which relates input states to possible output states. We call a sequence of statements $\tau \in$ **Stmt*** a *trace*, and extend the semantics in a straightforward way:

Definition 1 (Trace Semantics, Infeasibility). *The* semantics *of a trace $\tau \in \mathbf{Stmt}^*$ is recursively defined as*

$$[\![\varepsilon]\!] = id \qquad\qquad [\![st.\tau]\!] = [\![st]\!] \circ [\![\tau]\!]$$

We call τ infeasible *iff $[\![\tau]\!] = \emptyset$.*

The semantics of a trace is hence exactly the set of all pairs of program states (s, s') such that, starting from s, τ can be executed in its entirety, and can (depending on nondeterministic choices) reach the state s'. If no such pair exists, it follows that data constraints prevent the execution of the trace: It is infeasible.

A *state assertion* is a logical formula φ over variables in **Var**. We write $s \models \varphi$ to signify that program state s satisfies the state assertion φ. A valid Hoare triple $\{\varphi\}\, st\, \{\psi\}$ consists of state assertions φ, ψ and a statement st, such that for each pair $(s, s') \in [\![st]\!]$ it holds that $s \models \varphi$ implies $s' \models \psi$. An *infeasibility proof* for a trace $\tau = st_1 \dots st_n$ is a sequence of state assertions $\varphi_0 \dots \varphi_n$ such that $\varphi_0 = \mathit{true}$, $\varphi_n = \mathit{false}$ and $\{\varphi_i\}\, st_{i+1}\, \{\varphi_{i+1}\}$ is a valid Hoare triple for $i \in \{0, \dots, n-1\}$. As the name implies, if there exists an infeasibility proof for a trace τ, then τ is infeasible. In a sense, an infeasibility proof consists of data constraints blocking the execution of the trace τ.

Our algorithm considers bounded Petri nets $\mathcal{N}$ over a finite alphabet $\Sigma \subseteq \mathbf{Stmt}$. We use the term *Petri net* when we want to stress that it is only viewed as a language acceptor, ignoring the semantics of statements. By contrast, the term *Petri program* refers to the infinite-state program, which is derived by assigning semantics to the alphabet statements. The Petri net represents the control flow of this program. In fact, it could be called the program's *control flow Petri net*, in analogy to *control flow graphs* for sequential programs.

4.2 Verification Problem

In addition to the control flow, a Petri net $\mathcal{N}$ also encodes its correctness specification. This is achieved by the accepting places of the Petri net, which represent *error locations*, i.e., locations that should not be reached by any execution of the corresponding Petri program. In the program text from which the net is derived, these error locations are typically expressed as assert statements.

Example 2 (Specifications). For instance, the net in Fig. 3b encodes the program with the assert statement `assert x==42`. If this assert condition is *violated*, i.e., $x \neq 42$, the error location ℓ_{err} is reached.

An accepted trace $\tau \in L(\mathcal{N})$ is thus a trace for which at least one thread would reach an error location, provided the trace actually has a corresponding execution in the Petri program. We thus call these traces *error traces*. It now becomes clear that the verification task must consist of showing that no error trace has a corresponding execution in the Petri program, i.e., all error traces are *infeasible*.

Example 3 (Infeasible and Error Traces). Consider again the example shown in Fig. 2b. Here, the trace `x:=0` `x:=x+2` `x:=x*2` `x<0` is an error trace. It is however infeasible, as there is no output state s of `x:=0` `x:=x+2` `x:=x*2` which can execute `x<0`.

Conversely, the trace `x:=0` `x:=x+1` `x:=x+2` of Fig. 2b is feasible, its semantics contains for instance the pair of states $(\{x \mapsto 17\}, \{x \mapsto 3\})$. It is however not an error trace.

While the *language* $L(\mathcal{N})$ accepted by the Petri net represents the Petri program's control flow aspects, the *semantic relation* additionally takes into account the program's data and summarizes the program's semantics:

Definition 4 (Semantic Relation). *Let $\mathcal{N}$ be a Petri program. The* semantic relation $[\![\mathcal{N}]\!]$ *of $\mathcal{N}$ is defined as*

$$[\![\mathcal{N}]\!] := \bigcup_{\tau \in L(\mathcal{N})} [\![\tau]\!]$$

We conclude the section by formally stating the verification task:

Definition 5 (Petri Program Correctness). *A Petri program $\mathcal{N}$ is* correct *if and only if its semantic relation is empty:*

$$[\![\mathcal{N}]\!] = \emptyset$$

i.e., if and only if all error traces are infeasible.

5 Verification Algorithm

We now describe our verification algorithm. It is an adaptation of the *Trace Abstraction* approach [16] to Petri programs.

As defined above, the verification algorithm must determine if the semantic relation of a given Petri program $\mathcal{N}$ is empty, i.e., $[\![\mathcal{N}]\!] = \emptyset$. The simplest case where this holds is the case where the Petri net recognizes the empty language, i.e., $L(\mathcal{N}) = \emptyset$. This holds if and only if no accepting marking is reachable in $\mathcal{N}$. This reachability problem can be solved efficiently using the algorithm for the construction of *complete finite prefixes of the unfolding of a Petri net* proposed by McMillan [9,20]. This prefix is a finite initial part of the unfolding which contains full information about the reachable markings of the Petri net.

Our algorithm aims to reduce every verification problem to the simple case $L(\mathcal{N}) = \emptyset$, and thus to purely automata-theoretic reasoning. It does so by iteratively transforming the Petri program $\mathcal{N}$ to a Petri program $\mathcal{N}'$ that is equivalent, i.e., it has the same semantic relation: $[\![\mathcal{N}]\!] = [\![\mathcal{N}']\!]$. The transformed Petri net $\mathcal{N}'$ accepts only a subset of the traces accepted by $\mathcal{N}$, i.e., $L(\mathcal{N}') \subseteq L(\mathcal{N})$. If the algorithm eventually reaches a Petri net $\mathcal{N}'$ with $L(\mathcal{N}') = \emptyset$, as determined by the unfolding algorithm, then it holds that $[\![\mathcal{N}]\!] = [\![\mathcal{N}']\!] = \emptyset$, and the original Petri program $\mathcal{N}$ is correct.

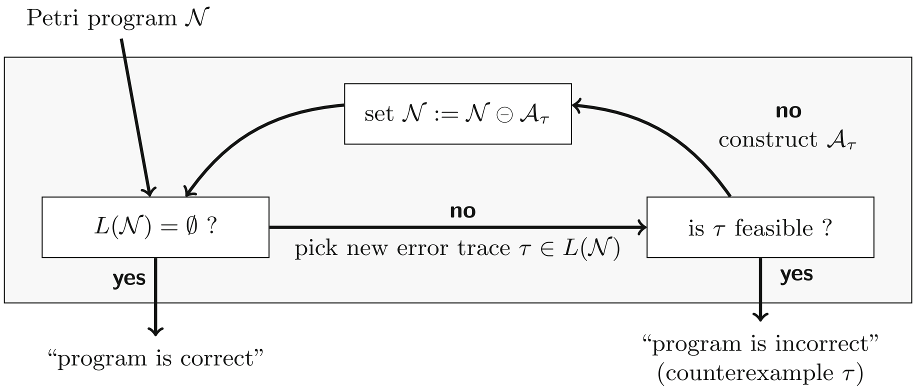

Petri program $\mathcal{N}$
set $\mathcal{N} := \mathcal{N} \ominus \mathcal{A}_\tau$
no
construct $\mathcal{A}_\tau$
$L(\mathcal{N}) = \emptyset$?
no
pick new error trace $\tau \in L(\mathcal{N})$
is τ feasible ?
yes
yes
"program is correct"
"program is incorrect" (counterexample τ)

Fig. 4. Trace Abstraction CEGAR loop.

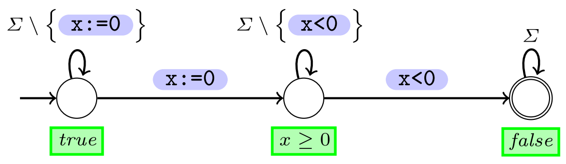
In order to achieve this iterative refinement, the algorithm employs a counterexample guided abstraction refinement (CEGAR) loop, illustrated in Fig. 4. In each iteration, the complete finite prefix of the unfolding of $\mathcal{N}$ is constructed in order to search for an error trace τ accepted by the current Petri net $\mathcal{N}$, the (possibly spurious) counterexample. Our algorithm then checks this counterexample for feasibility using an SMT solver. If the counterexample is feasible, i.e., non-spurious, then the program is incorrect and the verification is stopped. If τ is infeasible on the other hand, the algorithm constructs a finite automaton $\mathcal{A}_\tau$ that accepts at least τ and possibly infinitely many other infeasible traces. The refined Petri net is then constructed as the difference $\mathcal{N} \ominus \mathcal{A}_\tau$ of the current $\mathcal{N}$ and $\mathcal{A}_\tau$. We will describe the automata-theoretic difference operation $\ominus$ in Sect. 6. For the moment, suffice it to say that this difference operation takes as input a Petri net $\mathcal{N}$ and a finite automaton $\mathcal{A}$. It then constructs a version of the Petri net with additional synchronization, as seen in Sect. 2. The resulting Petri net satisfies $L(\mathcal{N} \ominus \mathcal{A}) = L(\mathcal{N}) \setminus L(\mathcal{A})$.

We now discuss how to construct the automaton $\mathcal{A}_\tau$. This automaton extracts the data constraints from an infeasibility proof of the trace τ and generalizes them to other traces. To this end, we introduce the following class of automata:

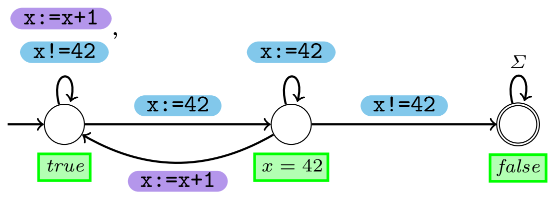
Definition 6 (Floyd/Hoare-Automata). *A* Floyd/Hoare-automaton *is a finite automaton $\mathcal{A} = (\Sigma, Q, \delta, q_{\mathsf{init}}, Q_{\mathsf{acc}})$ over the alphabet of program statements Σ, such that there exists a mapping β that assigns each automaton state $q \in Q$ a state assertion $\beta(q)$ with*

- *$\beta(q_{\mathsf{init}}) = true$,*
- *if $(q, st, q') \in \delta$, then $\{\beta(q)\}\, st\, \{\beta(q')\}$ is a valid Hoare triple,*
- *and for all $q \in Q_{\mathsf{acc}}$, $\beta(q) = false$.*

For each trace τ accepted by a Floyd/Hoare-automaton $\mathcal{A}$, there exists an infeasibility proof, given by application of β to the accepting run of states. Hence $\mathcal{A}$


$\Sigma \setminus \{$ x:=0 $\}$
$\Sigma \setminus \{$ x<0 $\}$
Σ
x:=0
x<0
true
$x \geq 0$
false

(a) Floyd/Hoare-automaton for a trace of Fig. 2b.



(b) Floyd/Hoare-automaton for a trace of Fig. 3b.

Fig. 5. Examples of deterministic total Floyd/Hoare-automata for the examples from Sect. 2.

can only accept infeasible traces. For details on how to construct a Floyd/Hoare-automaton $\mathcal{A}_\tau$ from an infeasible trace τ, we refer the reader to Heizmann et al. [16]. Note that in particular, one can always construct a deterministic total Floyd/Hoare-automaton, assuming the set of state assertions used to label states is closed under conjunctions.

Example 7 (Floyd/Hoare-Automata). Consider again an error trace from Fig. 2b, for instance `x:=0` `x:=x+1` `x:=x+2` `x:=x*2` `x:=x*1` `x<0`. A possible infeasibility proof is $\mathit{true}, x \geq 0, x \geq 0, x \geq 0, x \geq 0, x \geq 0, \mathit{false}$. A Floyd/Hoare-automaton corresponding to this infeasibility proof is shown in Fig. 5a. Subtraction of this Floyd/Hoare-automaton from the Petri net yields the Petri net shown in Fig. 2c.

Similarly, the error trace `x:=42` `x!=42` from Fig. 3b is proven infeasible by the sequence $\mathit{true}, x = 42, \mathit{false}$. A corresponding Floyd/Hoare-automaton is given in Fig. 5b. The Petri net shown in Fig. 3c represents the difference of the original net and this Floyd/Hoare-automaton.

After presenting the difference operation in Sect. 6, we will discuss in Sect. 7 how its usage combined with Floyd/Hoare-automata achieves the additional synchronization using data constraints. We conclude this section with a discussion of soundness of our approach. We begin by showing that the subtraction of a Floyd-Hoare automaton does not modify the Petri program's semantics.

Lemma 8 (Semantics-Preserving Refinement). *Let $\mathcal{A}_\tau$ be an automaton accepting only infeasible traces, and let $\mathcal{N}' = \mathcal{N} \ominus \mathcal{A}_\tau$. Then $\mathcal{N}$ is equivalent to $\mathcal{N}'$, i.e., $[\![\mathcal{N}]\!] = [\![\mathcal{N}']\!]$.*

Proof. Since $L(\mathcal{N}') \subseteq L(\mathcal{N})$, it follows that $[\![\mathcal{N}']\!] \subseteq [\![\mathcal{N}]\!]$. On the other hand, $L(\mathcal{N}) \subseteq L(\mathcal{N}') \cup L(\mathcal{A}_\tau)$. It follows that

$$[\![\mathcal{N}]\!] \subseteq [\![\mathcal{N}']\!] \cup \bigcup_{\tau \in L(\mathcal{A}_\tau)} [\![\tau]\!] = [\![\mathcal{N}']\!] \cup \bigcup_{\tau \in L(\mathcal{A}_\tau)} \emptyset = [\![\mathcal{N}']\!]$$

Thus it holds that $[\![\mathcal{N}]\!] = [\![\mathcal{N}']\!]$.

Intuitively, the argument for the equivalence of $\mathcal{N}$ and $\mathcal{N}'$ is this: We only remove traces $\tau \in L(\mathcal{A}_\tau)$ which are accepted by the Floyd/Hoare-automaton $\mathcal{A}_\tau$. But then such traces τ must be infeasible, i.e., there do not exist corresponding executions of the Petri program. Hence we only remove traces that are artefacts of the finite-state abstraction given by the Petri net, we never remove actual feasible program traces. In other words, we *refine* the abstraction $\mathcal{N}$ to the equivalent, but strictly less coarse abstraction $\mathcal{N}'$. We arrive at the soundness result for our algorithm:

Theorem 9 (Soundness). *Our verification algorithm is sound, i.e., whenever it concludes that a given input Petri program $\mathcal{N}$ is correct, then $[\![\mathcal{N}]\!] = \emptyset$.*

Proof. The algorithm iteratively transforms $\mathcal{N}$ to some Petri program $\mathcal{N}'$. By repeated application of Lemma 8, we have that $[\![\mathcal{N}]\!] = [\![\mathcal{N}']\!]$. The algorithm concludes correctness only if $L(\mathcal{N}') = \emptyset$ (by soundness of McMillan's unfolding algorithm [9,20]), and hence $[\![\mathcal{N}]\!] = [\![\mathcal{N}']\!] = \emptyset$.

6 Difference Operation

In this section we present a difference operation $\mathcal{N} \ominus \mathcal{A}$ for a Petri net $\mathcal{N}$ and a finite automaton $\mathcal{A}$. This difference operation implements the addition of synchronization discussed in Sect. 2, and is used by our verification algorithm as presented in Sect. 5. We give a purely automata theoretic presentation of this operation here, and we discuss in Sect. 7 how this operation implements the addition of synchronization constraints.

The inputs of our operation are a bounded Petri net $\mathcal{N} = (\Sigma, P, T, F, m_0, \lambda, P_{\mathsf{fin}})$ and a *deterministic total* finite automaton $\mathcal{A} = (\Sigma, Q, \delta, q_0, Q_{\mathsf{acc}})$ over the same alphabet which satisfies the property

$$L(\mathcal{A}) = L(\mathcal{A}) \circ \Sigma^*$$

i.e., the language of $\mathcal{A}$ is closed under concatenation with Σ^*. We call the Petri net $\mathcal{N}$ the *minuend* of the operation and we call the finite automaton $\mathcal{A}$ the *subtrahend* of the operation.

The basic idea of the construction is to run the Petri net and the DFA in parallel and to let the result block as soon as the DFA is going to enter a accepting state. The basic construction rule is illustrated in Fig. 6: For each Petri net transition t with predecessor places $p_1, \dots p_n$ and successor places $p'_1, \dots p'_m$, and for each edge in the finite automaton that has predecessor q, successor q'

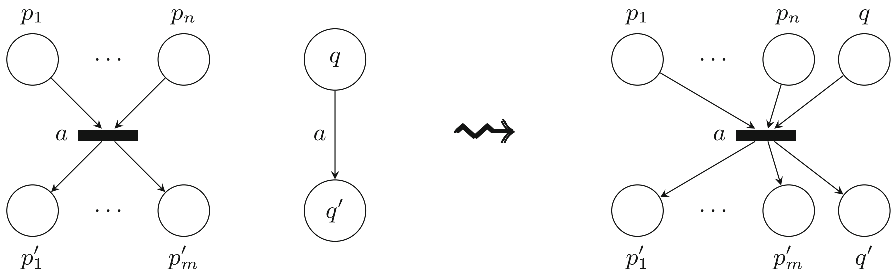


Fig. 6. Basic construction rule for the difference operation whose minuend is a Petri net, whose subtrahend is a deterministic total finite automaton and whose result is a Petri net.

and is labeled by $\lambda(t)$, we add a transition that is labeled by labeled by $\lambda(t)$ and has predecessor places are $p_1, \dots p_n, q$ and successor places $p'_1, \dots p'_m, q'$. There are however two exceptions to this basic construction rule.

E1: If the successor state q' is an accepting state, the transition must never fire. Hence we add a special *blocking place* as predecessor, which can never have a token.

E2: If a letter $a \in \Sigma$ occurs only in self-loops of the DFA, then we just copy the Petri net transitions that are labeled by a without adding an additional predecessor or successor.

The exception E1 ensures that words of $L(\mathcal{A})$ are not accepted by the result. The exception E2 is an optimization that reduces the number of transitions and the elements of the flow relation. While this optimization is not directly necessary for the correctness of the operation, we will discuss in Sect. 7 why it is crucial to our approach. In order to implement exception E2, we define the subset $\Sigma_{\mathsf{looper}} \subseteq \Sigma$ that consists of all letters $a \in \Sigma$ that occur only in self-loops of the subtrahend $\mathcal{A}$.

$$\Sigma_{\mathsf{looper}} := \{a \in \Sigma \mid \delta(q, a) = q \text{ for all } q \in Q\}$$

We define the result of the synchronization operation $\mathcal{N}' := \mathcal{N} \ominus \mathcal{A}$ formally as

$$\mathcal{N}' := (\Sigma, P', T', F', m'_0, \lambda', P'_{\mathsf{fin}})$$

The set of places is the disjoint union of the minuend's places, the subtrahend's states and one auxiliary place.

$$P' := P \,\dot\cup\, Q \,\dot\cup\, \{p_{\mathsf{block}}\}$$

The set of transitions and the flow relation is defined according to the construction rule of Fig. 6 and the exceptions E1 and E2.

$$\begin{aligned} T' := \quad & \{t \mid t \in T, \lambda(t) \in \Sigma_{\mathsf{looper}}\} \\ & \cup \{(q,t,q') \mid t \in T, \lambda(t) \notin \Sigma_{\mathsf{looper}}, q \in Q, \delta(q,\lambda(t)) = q'\} \\ F' := \quad & \{(p,t'),(t',p') \mid t' \in T, (p,t) \in F, (t,p') \in F\} \\ & \cup \{(q,t'),(t',q') \mid t' = (q,t,q'), q \in Q, t \in T, q' \in Q\} \\ & \cup \{(p,t'),(t',p') \mid t' = (q,t,q'), (p,t) \in F, (t,p') \in F\} \\ & \cup \{(p_{\mathsf{block}},t') \mid t' = (q,t,q'), q' \in Q_{\mathsf{acc}}\} \end{aligned}$$

Transition labels are copied from the minuend.

$$\lambda(t') := \begin{cases} \lambda(t') & \text{if } t' \in T \\ \lambda(t) & \text{if } t' = (q,t,q') \text{ for some } q \in Q, t \in T, q' \in Q \end{cases}$$

The initial marking is the disjoint union of the minuend's initial marking and the initial state of the subtrahend.

$$m_0' := m_0 \,\dot\cup\, \{q_0\}$$

The set of accepting places is the minuend's set of accepting places.

$$P'_{\mathsf{fin}} := P_{\mathsf{fin}}$$

We show that this operation does indeed implement the language-theoretic difference between the given Petri net and the automaton:

Theorem 10. *Given a Petri net $\mathcal{N}$ and a DFA $\mathcal{A}$. If $\mathcal{A}$ is total and closed under concatenation with Σ^*, then the Petri net $\mathcal{N} \ominus \mathcal{A}$ recognizes the set theoretic difference of $L(\mathcal{N})$ and $L(\mathcal{A})$, i.e.,*

$$L(\mathcal{N} \ominus \mathcal{A}) = L(\mathcal{N}) \setminus L(\mathcal{A}).$$

Proof. Let $\mathcal{N} = (\Sigma, P, T, F, m_0, \lambda, P_{\mathsf{acc}})$, $\mathcal{A} = (\Sigma, Q, \delta, q_0, Q_{\mathsf{acc}})$, and $\mathcal{N}' = \mathcal{N} \ominus \mathcal{A} = (\Sigma, P', T', F', m_0', \lambda', P'_{\mathsf{acc}})$. Let $a_1 \ldots a_n \in \Sigma^*$ be a word. We prove by induction over the length n the following: The sequence $m_0 \cup \{q_0\} \rhd_{t_1'} m_1 \cup \{q_1\} \rhd_{t_2'} \ldots \rhd_{t_n'} m_n \cup \{q_n\}$ is a firing sequence of $\mathcal{N}'$ iff $m_0 \rhd_{t_1} m_1 \rhd_{t_2} \ldots \rhd_{t_n} m_n$ is a firing sequence of $\mathcal{N}$ and $q_0, q_1, \ldots, q_n$ is a run of $\mathcal{A}$ such that no q_i is an accepting state, where $t_i' = t_i$ if $a_i \in \Sigma_{\mathsf{looper}}$ and $t_i' = (q_{i-1}, t_i, q_i)$ if $a_i \notin \Sigma_{\mathsf{looper}}$. In the induction step, we use that the DFA $\mathcal{A}$ is total and that our auxiliary place p_{block} ensures that the firing sequence of $\mathcal{N}'$ cannot contain an accepting state of $\mathcal{A}$. Since $\mathcal{A}$ is total it cannot block and since it is deterministic and closed under concatenation with Σ^* is can never leave the set of accepting states once it entered an accepting state. Together with the fact that the accepting places of $\mathcal{N}'$ are the accepting places of $\mathcal{N}$ we conclude that $a_1 \ldots a_n \in L(\mathcal{N}')$ iff $a_1 \ldots a_n \in L(\mathcal{N})$ and $a_1 \ldots a_n \notin L(\mathcal{A})$.

7 Discussion

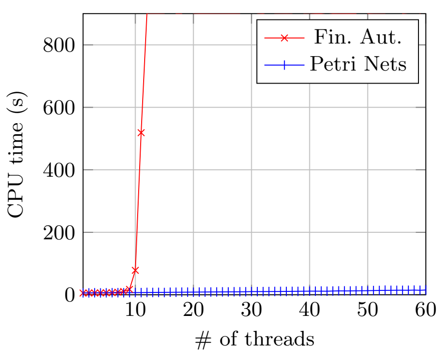
The approach we have presented avoids eager synchronization: It does not initially represent the many different interleavings of a concurrent program explicitly, and hence avoids the associated state explosion. Instead, synchronization is added lazily, only where it is needed: We derive data constraints from the analysis of infeasible interleavings, represented as a finite automaton. Our difference operation then uses this automaton to add synchronization to the Petri net. Here too we carefully avoid unnecessary synchronization.

Example 11 (Synchronization based on data constraints). Let us discuss the synchronization based on data constraints on our example programs. Consider the Petri net in Fig. 2c, the result of our difference construction applied to the Petri net in Fig. 2b and the Floyd/Hoare-automaton in Fig. 5a. Observe that there is no synchronization between the transitions labeled `x:=x+1`, `x:=x*1`, `x:=x+2` and `x:=x*2`. While the ordering between these statements does have an impact on the behaviour of the program (in particular, on the final value of `x`), it is irrelevant to the data constraint $x \geq 0$, which prevents the program from reaching the error place. Correspondingly, these statements only occur as self-loops in the Floyd/Hoare-automaton, and are thus not synchronized by our difference operation (exception E2). By contrast, the statement `x:=0` establishes the data constraint $x \geq 0$, and `x<0` contradicts it. Hence, the transitions labeled with these statements are modified to have additional predecessor and successor places corresponding to the automaton states. If the transition labeled `x<0` was to fire, the token would move to the place labeled *false*. This place corresponds to an accepting state of the Floyd/Hoare-automaton, or in other words, a violation of the data constraints. Hence the transition must not fire: It requires a token from the blocking place, which never has a token. In summary, we only synchronize the initialization statement and the check of the postcondition. The two threads however remain completely unsynchronized.

Compare this to our second example program, and to the difference of Fig. 3b and Fig. 5b, as shown in Fig. 3c. Here, synchronization between all three statements has been introduced: Each transition has as predecessor and as successor a place corresponding to an automaton state, resp. a data constraint. The reason for this lies in the fact that none of the statements is irrelevant to the data constraints. Hence synchronization is necessary to keep track of these constraints. The statement `x:=42` establishes the data constraint $x = 42$, and hence moves a token from the *true* place to the place labeled $x = 42$. The statement `x!=42` contradicts the data constraint $x = 42$, and hence we have a transition labeled with this statement, which takes a token out of the $x = 42$ place and puts it in the *false* place. However, we again prevent this transition from firing through the addition of a blocking place. Note that we have a second transition with the same statement: If the constraint $x = 42$ has not been established, the statement may execute without contradicting the trivial constraint *true*. No such second transition is necessary for the statement `x:=42`, as it is not possible to reach the data constraint $x = 42$ and then execute the statement (again). Finally, the statement `x:=x+1` is also synchronized: While it neither establishes nor contradicts the data constraint, this synchronization is necessary as it can invalidate the constraint. Here too we have two copies of the transition: If no constraint on the data has been established (a token is in the *true* place), the statement can execute and establishes no new data constraint (the token is put back into the *true* place). On the other hand, if the data constraint $x = 42$ holds (i.e., there is a token in the place labeled with this constraint), then `x:=x+1` invalidates the constraint (and moves the token into the *true* place). As a result of the added synchronization, only one (out of three) interleavings of the statements remains possible in the Petri net.

Optimization E2. Through the use of McMillan's unfolding technique, we are able to check emptiness of the refined Petri net resulting from our difference operation. The efficiency of this check and our whole approach relies crucially on the optimization E2 in the definition of the difference operation. Unfoldings do not explicitly consider the many different orderings between concurrent transitions, i.e., transitions t_1, t_2 that are both enabled in a reachable marking m and have disjoint sets of predecessors. The ability to preserve the concurrency of such transitions is the source of the efficiency of the unfolding technique. Without exception E2, the difference Petri net would not have any concurrent transitions: If the marking m can be reached from the initial marking through a firing sequence labeled with a word w, then by Theorem 10, m contains exactly one place corresponding to a state q of the finite automaton, namely the state that the automaton reaches after reading w. Without exception E2, q would be a predecessor place for both transitions t_1 and t_2, and hence the transitions would not be concurrent. Since unfoldings explicitly consider the ordering between non-concurrent transitions, they would suffer the same exponential explosion as naive sequentialization. Thus, our difference operation is specifically designed to optimize for the application of the unfolding technique.

Scalability. The result is a verification algorithm that, for many concurrent programs, is significantly more efficient than classical Trace Abstraction based on sequentialization. We demonstrate this efficiency improvement using our example concurrent program schema from Fig. 1. To this effect, we analyzed instances of this program schema for up to 60 threads, both with classical Trace Abstraction and with the method presented here. We ran both analyses on a machine with an AMD EPYC 7351P 16-Core CPU 2.4 GHz and 128 GB RAM running Linux 5.8.12 und Java 1.8.0_202 64bit, and monitored them using the benchmarking tool benchexec [4]. The results can be seen in Fig. 7. The classical, automata-based Trace Abstraction (shown in red) falls victim to the state explosion problem, and reaches the timeout (15 min) for 12 threads or more. On the other hand, our approach (shown in blue) scales much better, and can analyse even the 60-thread instance in approximately 15 s. Similarly, the memory consumption of classical Trace Abstraction explodes quickly, while the memory consumption of our approach scales well. The erratic memory consumption of the classical approach for more than 12 threads is due to the timeouts.



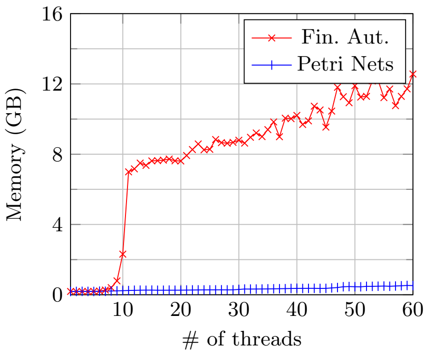


Fig. 7. Resources used in the analysis of instantiations of the program schema from Fig. 1 for up to 60 threads, with classical (automata-based) Trace Abstraction and with our Petri net-based approach.

8 Related Work

Partial Order Reduction (POR) is another technique used to deal with the complex control flow of concurrent systems, and has recently been applied for infinite-state program verification [5,6,12,13,22]. Closest to our approach are the works that combine POR with Trace Abstraction, such as the work by Cassez and Ziegler [5] as well as the works of Farzan and Vandikas [12,13]. Cassez and Ziegler apply a variant of POR, where two statements that do not write to common variables are independent. They apply this POR to the sequentialization of the concurrent program once, and then verify the resulting program using classical Trace Abstraction. Farzan and Vandikas on the other hand use a form of Büchi tree automata to represent an infinite range of reductions of the program and use an adaptation of Trace Abstraction to find a proof for one of these infinitely many reductions. We share the general idea of POR, namely to avoid explicitly representing many different interleavings. However, POR selects representative interleavings, while we consider all interleavings but represent them concisely. Furthermore, the works of Farzan and Vandikas in particular focus on proof simplicity . By contrast, our focus is on the combinatorial explosion of interleavings.

Bounded model checking (BMC) is among the most popular techniques for concurrent program verification. The basic idea in BMC is to search for a counterexample in executions whose length is bounded by some integer k. This problem can be efficiently reduced to a satisfiability problem, and can therefore be solved by SAT or SMT methods. BMC has the disadvantage of not being able to prove the absence of errors in general. There are many program verification tools based on bounded model checking, e.g., CBMC [1], DARTAGNAN [14], LAZY-CSEQ [18], and YOGAR-CBMC [23]. CBMC implements a bit-precise bounded model checking for C programs and uses POR to deal with the problem of interleavings. YOGAR-CBMC uses a scheduling constraint based abstraction refinement method for bounded model checking of concurrent programs. In order to obtain effective refinement constraints, two graph-based algorithms have been devised over the so-called Event Order Graph for counterexample validation and refinement generation. Lazy-CSeq translates a multi-threaded C program into a nondeterministic sequential C program that preserves reachability for all round-robin schedules with a given bound on the number of rounds and re-uses existing BMC tools as backends for the sequential verification problem.

Thread-Modular Abstraction Refinement [17] performs thread-modular assume-guarantee reasoning to overcome the challenge of the large number of interleavings of multithreaded programs. Thread modularity means that one explores the state space of one thread at a time, making assumptions about how the environment can interfere. This approach uses counterexample-guided predicate-abstraction refinement to overcome the challenge of the infinite state space.

Inductive data flow graphs [11] consist of data flow graphs with incorporated inductive assertions. They consider a set of dependencies between data operations in interleaved thread executions and generate the set of concurrent program traces which give rise to these dependencies. The approach first constructs an inductive data flow graph and then checks whether all program traces are represented.

Slab [8] is a certifying model checker for infinite-state concurrent systems. For a given transition system and a safety property it either delivers a counterexample or generates a certificate of system correctness in the form of an inductive verification diagram. Slab considers the control-flow constraints of a program as data constraints over program counter variables. Hence Slab can also abstract the control-flow of a program and does not have to build a product of CFGs initially. The abstraction is iteratively refined by predicates that are obtained from Craig interpolation.

9 Conclusion

We presented a verification approach for concurrent programs composed of a fixed number of threads over an infinite data domain. The contribution of the paper is to propose a solution to the two challenges raised by this verification problem: find a finite state abstraction for the concurrent program and deal with the problem of interleavings. Our solution is to use bounded Petri nets as finite state abstractions of the concurrent program. This enables us to apply algorithms based on unfoldings [9,20], i.e., algorithms that are used to analyze concurrent systems without falling victim to the problem of interleavings. Our algorithm for finding abstractions is based on the scheme of counterexample-guided abstraction refinement, specifically in the automata-based setting of Trace Abstraction [16]. We have shown that the automata-theoretic difference operation used in this setting can be implemented through the addition of (automatically generated) synchronization constraints to a Petri net.

References

1. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_9
2. Apt, K.R., de Boer, F.S., Olderog, E.R.: Verification of Sequential and Concurrent Programs. Springer Science & Business Media, Verlag (2009). https://doi.org/10.1007/978-1-84882-745-5
3. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 399–413. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_37
4. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2017). https://doi.org/10.1007/s10009-017-0469-y
5. Cassez, F., Ziegler, F.: Verification of concurrent programs using trace abstraction refinement. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 233–248. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_17
6. Chu, D.-H., Jaffar, J.: A framework to synergize partial order reduction with state interpolation. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 171–187. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13338-6_14
7. Donaldson, A., Kaiser, A., Kroening, D., Wahl, T.: Symmetry-aware predicate abstraction for shared-variable concurrent programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 356–371. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_28
8. Dräger, K., Kupriyanov, A., Finkbeiner, B., Wehrheim, H.: SLAB: a certifying model checker for infinite-state concurrent systems. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 271–274. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_22
9. Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan's unfolding algorithm. Formal Methods Syst. Des. **20**(3), 285–310 (2002). https://doi.org/10.1023/A:1014746130920
10. Farzan, A., Kincaid, Z.: Verification of parameterized concurrent programs by modular reasoning about data and control. ACM SIGPLAN Not. **47**(1), 297–308 (2012)
11. Farzan, A., Kincaid, Z., Podelski, A.: Inductive data flow graphs. ACM SIGPLAN Not. **48**(1), 129–142 (2013)
12. Farzan, A., Vandikas, A.: Automated hypersafety verification. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 200–218. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_11
13. Farzan, A., Vandikas, A.: Reductions for safety proofs. Proc. ACM Program. Lang. **4**(POPL), 13:1–13:28 (2020)
14. Gavrilenko, N., Ponce-de-León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC for weak memory models: relation analysis for compact SMT encodings. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 355–365. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_19
15. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 331–344. ACM (2011)

16. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 36–52. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_2
17. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread-modular abstraction refinement. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 262–274. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_27
18. Inverso, O., Trubiani, C.: Parallel and distributed bounded model checking of multi-threaded programs. In: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 202–216. ACM (2020)
19. Kahlon, V., Sankaranarayanan, S., Gupta, A.: Semantic reduction of thread interleavings in concurrent programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 124–138. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_12
20. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: von Bochmann, G., Probst, D.K. (eds.) CAV 1992. LNCS, vol. 663, pp. 164–177. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56496-9_14
21. Miné, A.: Static analysis of run-time errors in embedded critical parallel C programs. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 398–418. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19718-5_21
22. Wachter, B., Kroening, D., Ouaknine, J.: Verifying multi-threaded software with Impact. In: Formal Methods in Computer-Aided Design, pp. 210–217. IEEE (2013)
23. Yin, L., Dong, W., Liu, W., Wang, J.: On scheduling constraint abstraction for multi-threaded program verification. IEEE Trans. Softw. Eng. **46**(5), 549–565 (2020)