An SMT-based Approach to Coverability Analysis

Javier Esparza[1], Ruslán Ledesma-Garza[1], Rupak Majumdar[2], Philipp Meyer[1], and Filip Niksic[2]

[1] Technische Universität München
[2] Max Planck Institute for Software Systems (MPI-SWS)

Abstract. Model checkers based on Petri net coverability have been used successfully in recent years to verify safety properties of concurrent shared-memory or asynchronous message-passing software. We revisit a constraint approach to coverability based on classical Petri net analysis techniques. We show how to utilize an SMT solver to implement the constraint approach, and additionally, to generate an inductive invariant from a safety proof. We empirically evaluate our procedure on a large set of existing Petri net benchmarks. Even though our technique is incomplete, it can quickly discharge most of the safe instances. Additionally, the inductive invariants computed are usually orders of magnitude smaller than those produced by existing solvers.

1 Introduction

In recent years many papers have proposed and developed techniques for the verification of concurrent software [10,6,1,11,4]. In particular, model checkers based on Petri net coverability have been successfully applied. Petri nets are a simple and natural automata-like model for concurrent systems, and can model certain programs with an unbounded number of threads or thread creation. In a nutshell, the places of the net correspond to program locations, and the number of tokens in a place models the number of threads that are currently at that location. This point was first observed in [9], and later revisited in [3] and, more implicitly, in [10,6].

The problem whether at least one thread can reach a given program location (modelling some kind of error), naturally reduces to the *coverability problem* for Petri nets: given a net N and a marking M, decide whether some reachable marking of N *covers* M, i.e., puts at least as many tokens as M on each place. While the decidability and EXPSPACE-completeness of the coverability problem were settled long ago [12,17], new algorithmic ideas have been developed in recent years [8,7,21,11,13]. The techniques are based on forward or backward state-space exploration, which is accelerated in a number of ways in order to cope with the possibly infinite number of states.

In this paper we revisit an approach to the coverability problem based on classical Petri net analysis techniques: the marking equation and traps [16,18].

The marking equation is a system of linear constraints that can be easily derived from the net, and whose set of solutions overapproximates the set of reachable markings. This system can be supplemented with linear constraints specifying a set of unsafe markings, and solved using standard linear or integer programming. If the constraints are infeasible, then all reachable markings are safe. If not, then one can try different aproaches. In [5] a solution of the constraints is used to derive an additional constraint in the shape of a *trap*: a set of places that, loosely speaking, once marked cannot be "emptied"; the process can be iterated. More recently, in [22], Wimmel and Wolf propose to use the solution to guide a state space exploration searching for an unsafe marking; if the search fails, then information gathered during it is used to construct an additional constraint.

Constraint-based techniques, while known for a while, have always suffered from the absence of efficient decision procedures for linear arithmetic together with Boolean satisfiability. Profiting from recent advances in SMT-solving technology, we reimplement the technique of [5] on top of the Z3 SMT solver [2], and apply it to a large collection of benchmarks.

The technique is theoretically incomplete, i.e., the set of linear constraints derived from the marking equation and traps may be feasible even if all reachable markings are safe. Our first and surprising finding is that, despite this fact, the technique is powerful enough to prove safety of 96 out of a total of 115 safe benchmarks gathered from current research papers in concurrent software verification. In contrast, three different state-of-the-art tools for coverability proved only 61, 51, or 33 of these 115 cases! Moreover, and possibly due to the characteristics of the application domain, even the simplest version of the technique —based on the marking equation— is successful in 84 cases.

As a second contribution, and inspired by work on interpolation, we show that a dual version of the classical set of constraints, equivalent in expressive power, can be used not only to check safety, but to produce an inductive invariant. While some existing solvers based on state-space exploration can also produce such invariants, we show that inductive invariants obtained through our technique are usually orders of magnitude smaller. Additionally, while we can use the SMT solver iteratively to minimize the invariant, the tool almost always provides a minimal one at the first attempt.

Related Work. Our starting point was the work of Esparza and Melzer on extending the marking equation with trap conditions to gain a stronger method for proving safety of Petri nets [5]. We combined the constraint-based approach there with modern SMT solvers. Their focus on (integer) linear programming tools of the time enforced some limitations. First, while traps are naturally encoded using Boolean variables, [5] encoded traps and the marking equation together into a set of linear constraints. This encoding came at a practical cost: the encoding required (roughly) $n \times m$ constraints for a Petri net with n places and m transitions, whereas the natural Boolean encoding requires m constraints. Moreover, (I)LP solvers were not effective in searching large Boolean state spaces; our use of modern SAT techniques alleviates this problem. Second, (I)LP solvers used by [5] did not handle strict inequalities. Hence, the authors used additional tricks,

```
     procedure Process 1                    procedure Process 2
     begin                                  begin
       bit1 := false                          bit2 := false
       while true do                          while true do
p1:      bit1 := true                   q1:     bit2 := true
p2:      while bit2 do skip od          q2:     if bit1 then
p3:      (* critical section *)          q3:        bit2 := false
         bit1 := false                  q4:        while bit1 do skip od
       od                                        goto q1
     end                                     fi
                                      q5:    (* critical section *)
                                             bit2 := false
                                          od
                                        end
```

Fig. 1. Lamport's 1-bit algorithm for mutual exclusion [14].

such as posing the problem that includes a strict inequality as a minimization problem, with the goal of minimizing the involved expression, and testing if the minimal value equaled zero. Unfortunately, this trick led to numerical instabilities. All of these concerns vanish by using an SMT solver.

The marking equation is also the starting point of [22], but the strategies of this approach and ours are orthogonal: while we use the solutions of the marking equation to derive new constraints, [22] uses them to guide state space explorations that search for unsafe markings; new constraints are generated only if the searches fail.

In contrast to other recent techniques for coverability [7,11,13], our technique and the one of [22] are incomplete. However, in [22] Wimmel and Wolf obtain very good results for business process benchmarks, and in this paper we empirically demonstrate that our technique is effective for safe software verification benchmarks, often beating well-optimized state exploration approaches.

Our technique theoretically applies not only to coverability but also to *reachability*. It will be interesting to see whether the techniques can effectively verify reachability questions, e.g., arising from liveness verification [6].

2 Preliminaries

A *Petri net* is a tuple (P, T, F, m_0), where P is a set of *places*, T is a (disjoint) set of *transitions*, $F : (P \times T) \cup (T \times P) \to \{0, 1\}$ is the *flow function*, and $m_0 : P \to \mathbb{N}$ is the initial marking. For $x \in P \cup T$, the *pre-set* is ${}^\bullet x = \{y \in P \cup T \mid F(y, x) = 1\}$ and the *post-set* is $x^\bullet = \{y \in P \cup T \mid F(x, y) = 1\}$. We extend the pre- and post-set to a subset of $P \cup T$ as the union of the pre- and post-sets of its elements.

A *marking* of a Petri net is a function $m \colon P \to \mathbb{N}$, which describes the number of tokens $m(p)$ in each place $p \in P$. Assuming an enumeration $p_1, \ldots, p_n$ of P, we often identify m and the vector $(m(p_1), \ldots, m(p_n))$. For a subset $P' \subseteq P$ of places, we write $m(P') = \sum_{p \in P'} m(p)$.

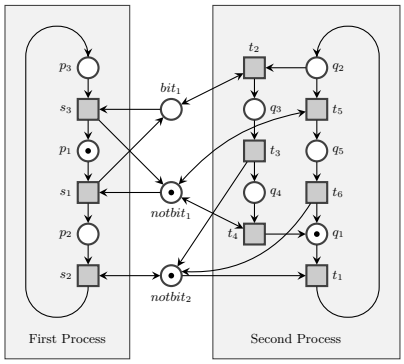

Fig. 2. Petri net for Lamport's 1-bit algorithm.

A transition $t \in T$ is *enabled at* m iff for all $p \in {}^\bullet t$, we have $m(p) \geq F(p,t)$. A transition t enabled at m may *fire*, yielding a new marking m' (denoted $m \xrightarrow{t} m'$), where $m'(p) = m(p) + F(t,p) - F(p,t)$. A sequence of transitions, $\sigma = t_1 t_2 \dots t_r$ is an *occurrence sequence* of N iff there exist markings $m_1, \dots, m_r$ such that $m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} m_2 \dots \xrightarrow{t_r} m_r$. The marking m_r is said to be *reachable* from m_0 by the occurrence of σ (denoted $m_0 \xrightarrow{\sigma} m_r$).

A *property* φ is a linear arithmetic constraint over the free variables P. The property φ holds on a marking m iff $m \models \varphi$. A Petri net N satisfies a property φ (denoted by $N \models \varphi$) iff for all reachable markings $m_0 \xrightarrow{\sigma} m$, we have $m \models \varphi$. A property φ is an *invariant* of N if it holds for every reachable marking. A property is inductive if whenever $m \models \varphi$ and $m \xrightarrow{t} m'$ for some $t \in T$ and marking m', we have $m' \models \varphi$.

Petri nets are represented graphically as follows: places and transitions are represented as circles and boxes, respectively. For $x, y \in P \cup T$, there is an arc leading from x to y iff $F(x,y) = 1$. As an example, consider Lamport's 1-bit algorithm for mutual exclusion [14], shown in Fig. 1. Fig. 2 shows a Petri net model for the code. The two grey blocks model the control flow of the two processes. For instance, the token in place p_1 models the current position of process 1 at program location p_1. The three places in the middle of the diagram model the current values of the variables. For instance, a token in place $notbit_1$ indicates that the variable bit_1 is currently set to *false*. The mutual exclusion property, which states that the two processes cannot be in the critical section at the same time, corresponds to the property that places p_3 and q_5 cannot both have a token at the same time.

3 Marking Equation

We now recall a well-known method, which we call SAFETY, that provides a sufficient condition for a given Petri net N to satisfy a property φ by reducing

the problem to checking satisfiability of a linear arithmetic formula. We illustrate the method on Lamport's 1-bit algorithm for mutual exclusion.

Before going into details, we state several conventions. For a Petri net $N = (P, T, F, m_0)$, we introduce a vector of $|P|$ variables M, and a vector of $|T|$ variables X. The vectors M and X will be used to represent the current marking and the number of occurrences of transitions in the occurrence sequence leading to the current marking, respectively. If a place or a transition is given a specific name, we use the same name for its associated variable. Given a place p, the intended meaning of a constraint like $p \geq 3$ is "at the current marking place p must have at least 3 tokens." Given a transition t, the intended meaning of a constraint like $t \leq 2$ is "in the occurrence sequence leading to the current marking, transition t must fire at most twice."

The key idea of the SAFETY method lies in the *marking equation*:

$$M = m_0 + CX\,,$$

where the incidence matrix C is a $|P| \times |T|$ matrix given by

$$C(p,t) = F(t,p) - F(p,t)\,.$$

For each place p, the marking equation contains a constraint that formulates a simple token conservation law: the number of tokens in p at the current marking is equal to the initial number of tokens $m_0(p)$, plus the number of tokens added by the input transitions of p, minus the number of tokens removed by the output transitions. So, for instance, in Lamport's algorithm the constraint for place $notbit_1$ is:

$$notbit_1 = 1 + s_3 + t_5 + t_4 - s_1 - t_4 - t_5 = 1 + s_3 - s_1\,.$$

We equip the marking equation with the non-negativity conditions, modeling that the number of tokens in a place, or the number of occurrences of a transition in an occurrence sequence cannot become negative. All together, we get the following set of *marking constraints*:

$$\mathcal{C}(P,T,F,m_0) :: \begin{cases} M = m_0 + CX & \text{marking equation} \\ M \geq 0 & \text{non-negativity conditions for places} \\ X \geq 0 & \text{non-negativity conditions for transitions} \end{cases}$$

Method SAFETY for checking that a property φ is invariant for a Petri net $N = (P, T, F, m_0)$ consists of checking for satisfiability of the constraints

$$\mathcal{C}(P,T,F,m_0) \wedge \neg\varphi(M)\,. \tag{1}$$

If the constraints are unsatisfiable, then no reachable marking violates φ. To see that this is true, consider the converse: If there exists an occurrence sequence $m_0 \xrightarrow{\sigma} m$ leading to a marking m that violates the property, then we can construct a valuation of the variables that assigns $m(p)$ to $M(p)$ for each place

p, and the number of occurrences of t in σ to $X(t)$ for each transition t. This valuation then satisfies the constraints.

The method does not work in the other direction: If the constraints (1) are satisfiable, we cannot conclude that the property φ is violated.

As an example, consider the Lamport's algorithm. SAFETY successfully proves the property "if process 1 is at location p_3, then $bit_1 = true$" by showing that $\mathcal{C}(P, T, F, m_0) \wedge p_3 \geq 1 \wedge bit_1 \neq 1$ is unsatisfiable. However, if we apply it to the mutual exclusion property, i.e., check for satisfiability of $\mathcal{C}(P, T, F, m_0) \wedge p_3 \geq 1 \wedge q_5 \geq 1$, we obtain a solution, but we cannot conclude that the mutual exclusion property does not hold.

Note that the marking constraints (1) are interpreted over integer variables. As usual in program analysis, one can solve the constraints over rationals to get an approximation of the method. Solving the constraints over rationals will become useful in Section 5.

4 Refining Marking Equations with Traps

Esparza and Melzer [5] strengthened SAFETY with additional *trap constraints*. A *trap* of a Petri net $N = (P, T, F, m_0)$ is a subset of places $Q \subseteq P$ satisfying the following condition for every transition $t \in T$: if t is an output transition of at least one place of Q, then it is also an input transition of at least one place of Q. Equivalently, Q is a trap if its set of output transitions is included in its set of input transitions, i.e., if $Q^\bullet \subseteq {}^\bullet Q$. Here we present a variant of Esparza's and Melzer's method that encodes traps using Boolean constraints. We call the new method SAFETYBYREFINEMENT.

The method SAFETYBYREFINEMENT is based on the following observation about traps. If Q is a trap and a marking m marks Q, i.e., $m(p) > 0$ for some $p \in Q$, then for each occurrence sequence σ and marking m' such that $m \xrightarrow{\sigma} m'$, we also have $m'(p') > 0$ for some $p' \in Q$. Indeed, by the trap property any transition removing tokens from places of Q also adds at least one token to some place of Q. So, while $m'(Q)$ can be smaller than $m(Q)$, it can never become 0. In particular, if a trap Q satisfies $m_0(Q) > 0$, then every reachable marking m satisfies $m(Q) > 0$ as well.

Since the above property must hold for any trap, we can restrict the constraints from method SAFETY as follows. First, we add an additional vector B of $|P|$ Boolean variables. These variables are used to encode traps: for $p \in P$, $B(p)$ is true if and only if place p is part of the trap. The following constraint specifies that B encodes a trap:

$$\mathit{trap}(B) ::= \bigwedge_{t \in T} \left[\bigvee_{p \in {}^\bullet t} B(p) \implies \bigvee_{p \in t^\bullet} B(p) \right] .$$

Next, we define a predicate $\mathit{mark}(m, B)$ that specifies marking m marks a trap:

$$\mathit{mark}(m, B) ::= \bigvee_{p \in P} B(p) \wedge (m(p) > 0) .$$

Finally, we conjoin the following constraint to the constraints (1):

$$\forall B : \mathit{trap}(B) \wedge \mathit{mark}(m_0, B) \implies \mathit{mark}(M, B)\,. \tag{2}$$

This constraint conceptually enumerates over all subsets of places, and ensures that if the subset forms a trap, and this trap is marked by the initial marking, then it is also marked by the current marking. Thus, markings violating the trap constraint are eliminated.

While the above constraint provides a refinement of the SAFETY method, it requires the SMT solver to reason with universally quantified variables. Instead of directly using universal quantifiers, we use a counterexample-guided heuristic [5,20] of adding trap constraints one-at-a-time in the following way.

If the set of constraints constructed so far (for instance, the set given by the method SAFETY) is feasible, the SMT solver delivers a model that assigns values to each place, corresponding to a potentially reachable marking m. We search for a trap P_m that violates the trap condition (2) for this specific model m. If we find such a trap, then we know that m is unreachable, and we can add the constraint $\sum_{p \in P_m} M(p) \geq 1$ to exclude all markings that violate this specific trap condition.

The search for P_m is a pure Boolean satisfiability question. We ask for an assignment to

$$\mathit{trap}(B) \wedge \mathit{mark}(m_0, B) \wedge \neg \mathit{mark}(m, B) \tag{3}$$

Notice that for a fixed marking m, the predicate $\mathit{mark}(m, B)$ simplifies to a Boolean predicate. Given a satisfying assignment b for this formula, we add the constraint

$$\sum_{\substack{p \in P \\ b(p) = true}} M(p) \geq 1 \tag{4}$$

to the current set of constraints to rule out solutions that do not satisfy this trap constraint. We iteratively add such constraints until either the constraints are unsatisfiable or the Boolean constraints (3) are unsatisfiable (i.e., no traps are found to invalidate the current solution).

This yields the method SAFETYBYREFINEMENT. It is still not complete [5]: one can find nets and unreachable markings that mark all traps of the net.

Let us apply the algorithm SAFETYBYREFINEMENT to Lamport's algorithm and the mutual exclusion property. Recall that the markings violating the property are those satisfying $p_3 \geq 1$ and $q_5 \geq 1$. SAFETY yields a satisfying assignment with $p_3 = bit_1 = q_5 = 1$, and $p = 0$ for all other places p, which corresponds to a potentially reachable marking m. We search for a trap marked at m_0 but not at m. To simplify the notation, we simply write p instead of $B(p)$. The

constraints derived from the trap property are:

$$\begin{array}{lcl} p_1 \vee notbit_1 & \Longrightarrow & p_2 \vee bit_1 \\ p_2 \vee notbit_2 & \Longrightarrow & p_3 \vee notbit_2 \\ p_3 \vee bit_1 & \Longrightarrow & p_1 \vee notbit_1 \end{array} \qquad \begin{array}{lcl} q_1 \vee notbit_2 & \Longrightarrow & q_2 \\ q_2 \vee bit_1 & \Longrightarrow & q_3 \vee bit_1 \\ q_3 & \Longrightarrow & q_4 \vee notbit_2 \\ q_4 \vee notbit_1 & \Longrightarrow & q_1 \vee notbit_1 \\ q_2 \vee notbit_1 & \Longrightarrow & q_5 \vee notbit_1 \\ q_5 & \Longrightarrow & q_1 \vee notbit_2 \end{array}$$

and the following constraints model that at least one of the places initially marked belongs to the trap, but none of the places marked at the satisfying assigment do:

$$p_1 \vee q_1 \vee notbit_1 \vee notbit_2 \qquad \neg p_3 \wedge \neg q_5 \wedge \neg bit_1$$

For this set of constraints we find the satisfying assignment that sets p_2, $notbit_1$, $notbit_2$, q_2, q_3 to *true* and all other variables to *false*. So this set of places is an initially marked trap, and so every reachable marking should put at least one token in it. Hence we can add the refinement constraint to marking constraints (1):

$$p_2 + q_2 + q_3 + notbit_1 + notbit_2 \geq 1\,.$$

On running the SMT solver again, we find the constraints are unsatisfiable, proving that the mutual exclusion property holds.

5 Constructing Invariants from Constraints

We now show that one can compute inductive invariants from the method SafetyByRefinement. That is, given a Petri net $N = (P, T, F, m_0)$ and a property φ, if SafetyByRefinement (over the rationals) can prove N satisfies φ, then in fact we can construct a linear inductive invariant that contains m_0 and does not intersect $\neg\varphi$. We call the new method InvariantByRefinement.

The key observation is to use a constraint system dual to the constraint system for SafetyByRefinement. We assume φ is a co-linear property, i.e., the negation $\neg\varphi$ is represented as the constraints:

$$\neg\varphi \;::\; AM \geq b$$

where A is a $k \times |P|$ matrix, and b is a $k \times 1$ vector, for some $k \geq 1$. Furthermore, we assume that there are $l \geq 0$ trap constraints (4), which are collected in matrix form $DM \geq 1$, for an $l \times |P|$ matrix D, and an $l \times 1$ vector of ones, denoted simply by 1. Consider the following primal system $\mathcal{S}$:

$\mathcal{C}(P, T, F, m_0)$	marking constraints
$AM \geq b$	negation of property φ
$DM \geq 1$	trap constraints

By transforming $\mathcal{S}$ into a suitable form and applying Farkas' Lemma [19], we get the following theorem.

Theorem 1. *The primal system $\mathcal{S}$ is unsatisfiable over the rational numbers if and only if the following dual system $\mathcal{S}'$ is satisfiable over the rational numbers.*

$$\begin{aligned} \lambda C &\leq 0 && \textit{inductivity constraint} \\ \lambda m_0 &< Y_1 b + Y_2 1 && \textit{safety constraint} \\ \lambda &\geq Y_1 A + Y_2 D && \textit{property constraint} \\ Y_1, Y_2 &\geq 0 && \textit{non-negativity constraint} \end{aligned}$$

Here λ, Y_1 and Y_2 are vectors of variables of size $1 \times |P|$, $1 \times k$ and $1 \times l$, respectively.

If the primal system $\mathcal{S}$ is unsatisfiable, we can take λ from a solution to $\mathcal{S}'$ and construct an inductive invariant:

$$I(M) \ ::= \ DM \geq 1 \wedge \lambda M \leq \lambda m_0 \, .$$

In order to show that $I(M)$ is an invariant, recall that for every reachable marking m there is a solution to $m = m_0 + CX$, with $X \geq 0$. Multiplying by λ and taking into account that λ is a solution to $\mathcal{S}'$, we get

$$\lambda m = \lambda m_0 + \lambda C X \leq \lambda m_0 \, .$$

Furthermore, every reachable marking satisfies the trap constraints $DM \geq 1$. On the other hand, a marking m that violates the property φ does not satisfy $I(M)$, for it either does not satisfy $DM \geq 1$, or both $Am \geq b$ and $Dm \geq 1$ hold. But in the latter case we have

$$\lambda m \geq (Y_1 A + Y_2 D)m = Y_1 Am + Y_2 Dm \geq Y_1 b + Y_2 1 > \lambda m_0 \, .$$

In order to show that $I(M)$ is inductive, we have to show that if $I(m)$ holds for some marking m (reachable or not), and $m \xrightarrow{t} m'$ for some transition t, then $I(m')$ holds as well. Indeed, in this case we have $m' = m + Ce_t$, where e_t is the unit vector with 1 in the t-th component and 0 elsewhere. Hence

$$\lambda m' = \lambda(m + Ce_t) = \lambda m + \lambda C e_t \leq \lambda m \leq \lambda m_0 \, ,$$

and furthermore, as m satisfies the trap constraints, m' also satisfies them.

So far, we have assumed that property φ is a co-linear property. However, it is easy to extend the method to the case when $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_r$, and each φ_i is a co-linear property. In that case, for each φ_i we invoke InvariantByRefinement to obtain an inductive invariant I_i. One can easily verify that $I_1 \wedge \ldots \wedge I_r$ is an inductive invariant with respect to φ.

Minimizing invariants. Note that the system $\mathcal{S}'$ from Theorem 1 may in general have many solutions, and each solution yields an inductive invariant. Solutions where λ has fewer non-zero components yield shorter inductive invariants $I(M)$, assuming terms in $I(M)$ with coefficient zero are left out. We can force the

Inductivity constraints

$$\begin{array}{rrrrl} -p_1 + p_2 & & + bit_1 - notbit_1 & \leq 0 \\ & -p_2 + p_3 & & \leq 0 \\ p_1 & & - p_3 - bit_1 + notbit_1 & \leq 0 \end{array} \qquad \begin{array}{rrrrrl} -q_1 + q_2 & & & & - notbit_2 & \leq 0 \\ & -q_2 + q_3 & & & & \leq 0 \\ & & -q_3 + q_4 & & + notbit_2 & \leq 0 \\ q_1 & & & -q_4 & & \leq 0 \\ & -q_2 & & & + q_5 & \leq 0 \\ q_1 & & & & -q_5 + notbit_2 & \leq 0 \end{array}$$

Safety constraint

$$p_1 + q_1 + notbit_1 + notbit_2 < target_1 + target_2 + trap_1$$

Property constraints

$$\begin{array}{llll} p_1 \geq 0 & q_1 \geq 0 & q_4 \geq 0 & bit_1 \geq 0 \\ p_2 \geq trap_1 & q_2 \geq trap_1 & q_5 \geq target_2 & notbit_1 \geq trap_1 \\ p_3 \geq target_1 & q_3 \geq trap_1 & & notbit_2 \geq trap_1 \end{array}$$

Non-negativity constraints

$$target_1, target_2, trap_1 \geq 0$$

Fig. 3. System of constraints $\mathcal{S}'$ for Lamport's algorithm and the mutual exclusion property. Here, $\lambda = (p_1\ p_2\ p_3\ q_1\ q_2\ q_3\ q_4\ q_5\ bit_1\ notbit_1\ notbit_2)$, $Y_1 = (target_1\ target_2)$ and $Y_2 = (trap_1)$.

number of non-zero components to be at most K by introducing a vector of $|P|$ variables Z, adding for each $p \in P$ constraints

$$\begin{aligned} \lambda(p) > 0 &\implies Z(p) = 1 \\ \lambda(p) = 0 &\implies Z(p) = 0 \end{aligned}$$

and adding a constraint $\sum_{p \in P} Z(p) \leq K$. By varying K, we can find a solution with the smallest number of non-zero components in λ.

Example. Consider again Lamport's algorithm and the mutual exclusion property. Recall that the negation of the property for this example is $p_3 \geq 1 \wedge q_5 \geq 1$, and the trap constraint is $p_2 + q_2 + q_3 + notbit_1 + notbit_2 \geq 1$. Fig. 3 shows the system of constraints $\mathcal{S}'$ for this example. A possible satisfying assignment sets q_1, q_4, and bit_1 to 0, p_2, p_3, and $target_1$ to 2, and all other variables to 1. The corresponding inductive invariant is:

$$\begin{aligned} I(M) \ ::= &\ (p_2 + q_2 + q_3 + notbit_1 + notbit_2 \geq 1) \wedge \\ &\ (p_1 + 2p_2 + 2p_3 + notbit_1 + notbit_2 + q_2 + q_3 + q_5 \leq 3)\,. \end{aligned}$$

If we add constraints that bound the number of non-zero components in λ to 7, the SMT solver finds a new solution, setting p_2, p_3, $notbit_1$, $notbit_2$, q_2, q_3,

$target_1$, $target_2$, and $trap_1$ to 1, and all other variables to 0. The corresponding inductive invariant for this solution is

$$\begin{aligned} I'(M) \ ::= & \ (p_2 + q_2 + notbit_1 + notbit_2 + q_3 \geq 1) \wedge \\ & (p_2 + p_3 + notbit_1 + notbit_2 + q_2 + q_3 + q_5 \leq 2)\,. \end{aligned}$$

6 Experimental Evaluation

We implemented our algorithms in a tool called *Petrinizer*. Petrinizer is implemented as a script on top of the Z3 SMT solver [2]. It takes as input coverability problem instances encoded in the MIST input format[3], and it runs one of the selected methods. We implemented all possible combinations of methods: with and without trap refinement, with rational and integer arithmetic, with and without invariant construction, with and without invariant minimization.

Our evaluation had two main goals. First, as the underlying methods are incomplete, we wanted to measure their success rate on standard benchmark sets. As a subgoal, we wanted to investigate the usefulness and necessity of traps, the benefit of using integer arithmetic over rational arithmetic, and the sizes of the constructed invariants. The second goal was to measure Petrinizer's performance and to compare it with state-of-the-art tools: IIC [13], BFC[4] [11], and MIST[5].

Benchmarks. For the inputs used in the experiments, we collected coverability problem instances originating from various sources. The collection contains 178 examples, out of which 115 are safe, and is organized into five example suites. The first suite is a collection of Petri net examples from the MIST toolkit. This suite contains a mixture of 29 examples, both safe and unsafe. It contains both real-world and artificially created examples. The second suite consists of 46 Petri nets that were used in the evaluation of BFC [11]. They originate from the analysis of concurrent C programs, and they are mostly unsafe. The third and the fourth suites come from the provenance analysis of messages in a medical system and a bug-tracking system [15]. The medical suite contains 12 safe examples, and the bug-tracking suite contains 41 examples, all safe except for one. The fifth suite contains 50 examples that come from the analysis of Erlang programs [4]. We generated them ourselves using an Erlang verification tool called Soter [4], from the example programs found on Soter's website[6]. Out of 50 examples in this suite, 38 are safe. This suite also contains the largest example in the collection,

[3] `https://github.com/pierreganty/mist`

[4] The most recent version of BFC at the time of writing the paper was 2.0. However, we noticed it sometimes reports inconsistent results, so we used version 1.0 instead. The tool can be obtained at `http://www.cprover.org/bfc/`.

[5] MIST consists of several methods, most of them based on EEC [8]. We used the abstraction refinement method that tries to minimize the number of places in the Petri net [7].

[6] `http://mjolnir.cs.ox.ac.uk/soter/`

Table 1. Safe examples that were successfully proved safe. Symbols $\mathbb{Q}$ and $\mathbb{Z}$ denote rational and integer numbers.

Suite	Safety/$\mathbb{Q}$	Safety/$\mathbb{Z}$	Ref./$\mathbb{Q}$	Ref./$\mathbb{Z}$	IIC	BFC	MIST	Total
MIST	14	14	20	20	**23**	21	19	23
BFC	**2**	**2**	**2**	**2**	**2**	**2**	**2**	2
Medical	4	4	4	4	9	**12**	10	12
Bug-tracking	**32**	**32**	**32**	**32**	0	0	0	40
Erlang	32	32	36	**38**	17	26	2	38
Total	84	84	94	**96**	51	61	33	115

with 66,950 places and 213,635 transitions. For our evaluation, only the 115 safe instances are interesting.

Rate of success on safe examples. As shown in Table 1, even with the weakest of the methods —safety based on marking equation over rationals— Petrinizer is able to prove safety for 84 out of 115 examples. Switching to integer arithmetic does not help: the number of examples proved safe remains 84. Using refinement via traps, Petrinizer proves safety for 94 examples. Switching to integer arithmetic in this case helps: Another two examples are proved safe, totaling 96 out of 115 examples. In contrast to these numbers, the most successful existing tool turned out to be BFC, proving safety for only 61 examples. Even though the methods these tools implement are theoretically complete, the tools themselves are limited by the time and space they can use.

Looking at the results accross different suites, we see that Petrinizer performed poorest on the medical suite, proving safety for only 4 out of 12 examples. On the other hand, on the bug-tracking suite, which was completely intractable for other tools, it proved safety for 32 out of 40 examples. Furthermore, using traps and integer arithmetic, Petrinizer successfuly proved safety for all safe Erlang examples. We find this result particularly surprising, as the original verification problems for these examples seem non-trivial.

Invariant sizes. We measure the size of inductive invariants produced by Petrinizer without minimization. We took the number of atomic (non-zero) terms appearing in an invariant's linear expressions as a measure of its size. When we relate sizes of invariants to number of places in the corresponding Petri net (top left graph in Fig. 4), we see that invariants are usually very succinct. As an example, the largest invariant had 814 atomic terms, and the corresponding Petri net, coming from the Erlang suite, had 4,763 places. For the largest Petri net, with 66,950 places, the constructed invariant had 339 atomic terms.

The added benefit of minimization is negligible: there are only four examples where the invariant was reduced, and the reduction was about 2-3%. Thus, invariant minimization does not pay off for these examples.

We also compared sizes of constructed invariants with sizes of invariants produced by IIC [13]. IIC's invariants are expressed as CNF formulas over atoms of the form $x < a$, for a variable x and a constant a. As a measure of size for these formulas, we took the number of atoms they contain. As the bottom left

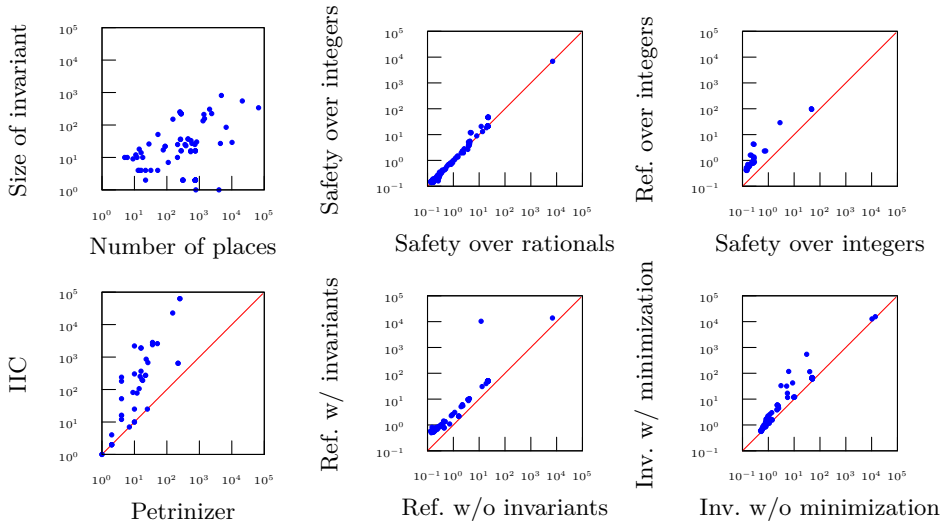

Fig. 4. Graph on the top left shows a relation of sizes of constructed invariants to the number of places in the corresponding Petri nets. Graph on the bottom left shows comparison in size of invariants produced by Petrinizer and IIC. Axes represent size on a logarithmic scale. Each dot represents one example. The four graphs in the center and on the right show time overhead of integer arithmetic, trap refinement, invariant construction and invariant minimization. Axes represent time in seconds on a logarithmic scale. Each dot represents execution time on one example. The graph on the top right only shows examples for which at least one trap appeared in the refinement. Similarly, the bottom center and bottom right graphs only show safe examples.

graph in Fig. 4 shows, when compared to IIC's invariants, Petrinizer's invariants are never larger, and are often orders of magnitude smaller.

Performance. To ensure accuracy and fairness, all experiments were performed on identical machines, equipped with Intel Xeon 2.66 GHz CPUs and 48 GB of memory, running Linux 3.2.48.1 in 64-bit mode. Execution time was limited to 100,000 seconds (27 hours, 46 minutes and 40 seconds), and memory to 2 GB.

Due to dissimilarities between the compared tools, selecting a fair measure of time was non-trivial. On the one hand, as Petrinizer communicates with Z3 via temporary files, it spends a considerable amount of time doing I/O operations. On the other hand, as BFC performs both a forward and a backward search, it naturally splits the work into two threads, and runs them in parallel on two CPU cores. In both cases, the actual elapsed time does not quite correspond to the amount of computational effort we wanted to measure. Therefore, for the measure of time we selected the *user time*, as reported by the *time* utility on Linux. User time measures the total CPU time spent executing the process and

Table 2. Mean and median times in seconds for each tool. We report times for safe examples, as well as for all examples. Memory-out cases were set to the timeout value of 100,000 s. Symbols $\mathbb{Q}$ and $\mathbb{Z}$ denote rational and integer numbers.

Method/tool	Safety/$\mathbb{Q}$	Safety/$\mathbb{Z}$	Ref./$\mathbb{Q}$	Ref./$\mathbb{Z}$	Safety+inv.	Safety+inv.min.
Mean (safe)	69.26	70.20	69.36	72.20	168.46	203.05
Median (safe)	2.45	2.23	2.35	3.81	3.70	4.03
Mean (all)	45.17	46.04	45.52	47.70	109.23	131.58
Median (all)	0.44	0.43	0.90	0.93	0.66	1.00

Method/tool	Ref.+inv.	Ref.+inv.min.	IIC	BFC	MIST
Mean (safe)	228.88	275.12	56954.09	47126.12	69196.77
Median (safe)	5.96	6.30	100000.00	1642.43	100000.00
Mean (all)	148.57	178.45	44089.93	31017.80	61586.56
Median (all)	1.37	1.94	138.00	0.77	100000.00

its children. In the case of Petrinizer, it excludes the I/O overhead, and in the case of BFC, it includes total CPU time spent on both CPU cores.

We report mean and median times measured for each tool in Table 2.

Time overhead of Petrinizer's methods. Before comparing Petrinizer with other tools, we analyze time overhead of integer arithmetic, trap refinement, invariant construction, and invariant minimization. The four graphs in the center and on the right in Fig. 4 summarize the results. The top central graph shows that the difference in performance between integer and rational arithmetic is negligible.

The top right graph in Fig. 4 shows that traps incur a significant overhead. This is not too surprising as, each time a trap is found, the main system has to be updated with a new trap constraint and solved again. Thus the actual overhead depends on the number of traps that appear during the refinement. In the experiments, there were 32 examples for refinement with integer arithmetic where traps appeared at least once. The maximal number of traps in a single example was 9. In the examples where traps appear once, we see a slowdown of 2-3×. In the extreme cases with 9 traps we see slowdowns of 10-16×.

In the case of invariant construction, as shown on the bottom central graph in Fig. 4, the overhead is more uniform and predictable. The reason is that constructing the invariant involves solving the dual form of the main system as many times as there are disjuncts in the property violation constraint. In most cases, the property violation constraint has one disjunct. A single example with many disjuncts, having 8989 of them, appears on the graph as an outlier.

In the case of invariant minimization, as the bottom right graph in Fig. 4 shows, time overhead is quite severe. The underlying data contains examples of slowdowns of up to 30×.

Comparison with other tools. The six graphs in Fig. 5 show the comparison of execution times for Petrinizer vs. IIC, BFC, and MIST. In the comparison, we used the refinement methods, both with and without invariant construction. In general, we observe that other tools outperform Petrinizer on small examples, an effect that can be explained by the overhead of starting script interpreters and Z3. However, on large examples Petrinizer consistently outperforms other tools.

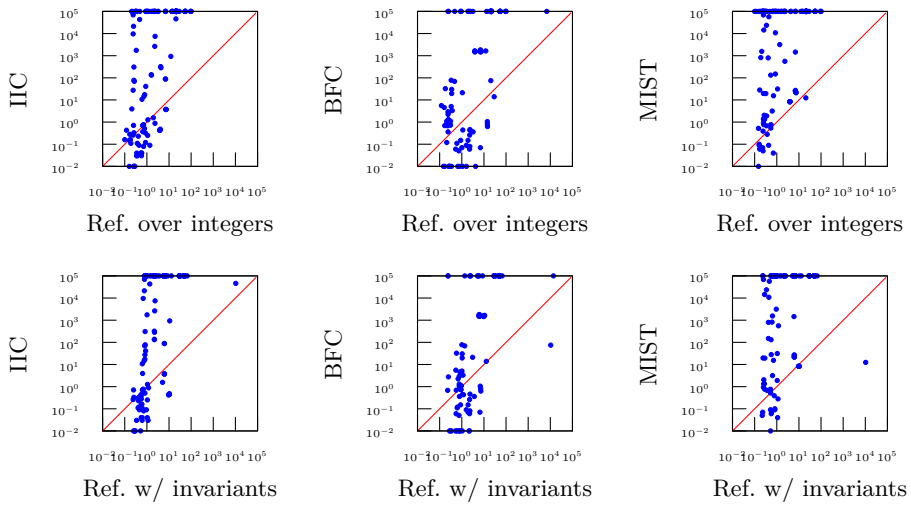

Fig. 5. Comparison of execution time for Petrinizer vs. IIC, BFC and MIST. Graphs in the top row show comparison in the case without invariant construction, and graphs in the bottom row show comparison in the case with invariant construction. Axes represent time in seconds on a logarithmic scale. Each dot represents execution time on one example.

Not only does it finish in all cases within the given time and memory constraints, it even finishes in under 100 seconds in all but two cases. The two cases are the large example from the Erlang suite, with 66,950 places and 213,635 transitions and, in the case of invariant construction, the example from the MIST suite, with 8989 disjuncts in the property violation constraint.

Conclusions. Marking equations and traps are classical techniques in Petri net theory, but have fallen out of favor in recent times in comparison with state-space traversal techniques in combination with abstractions or symbolic representations. Our experiments demonstrate that, when combined with the power of a modern SMT solver, these techniques can be surprisingly effective in finding proofs of correctness (inductive invariants) of common benchmark examples arising out of software verification.

Our results also suggest incorporating these techniques into existing tools as a cheap preprocessing step. A finer integration with these tools is conceivable, where a satisfying assignment to a system of constraints is used to guide the more sophisticated search, similar to [22].

Acknowledgements. We thank Emanuele D'Osualdo for help with the Soter tool. Ledesma-Garza was supported by the Collaborative Research Center 1480 "Program and Model Analysis" funded by the German Research Council.

References

1. A. Bouajjani and M. Emmi. Bounded phase analysis of message-passing programs. In *TACAS*, pages 451–465, 2012.
2. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
3. G. Delzanno, J.-F. Raskin, and L. Van Begin. Towards the automated verification of multithreaded Java programs. In *TACAS*, pages 173–187, 2002.
4. E. D'Osualdo, J. Kochems, and C.-H. L. Ong. Automatic verification of Erlang-style concurrency. In *Proceedings of the 20th Static Analysis Symposium*, SAS'13. Springer-Verlag, 2013.
5. J. Esparza and S. Melzer. Verification of safety properties using integer programming: Beyond the state equation. *Formal Methods in System Design*, 16(2):159–189, 2000.
6. P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.*, 34(1):6, 2012.
7. P. Ganty, J.-F. Raskin, and L. Van Begin. From many places to few: Automatic abstraction refinement for Petri nets. *Fundam. Inform.*, 88(3):275–305, 2008.
8. G. Geeraerts, J.-F. Raskin, and L. V. Begin. Expand, enlarge and check: New algorithms for the coverability problem of WSTS. *J. Comput. Syst. Sci.*, 72(1):180–203, 2006.
9. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM (JACM)*, 39(3):675–735, 1992.
10. A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV*, pages 645–659, 2010.
11. A. Kaiser, D. Kroening, and T. Wahl. Efficient coverability analysis by proof minimization. In *CONCUR*, pages 500–515, 2012.
12. R. Karp and R. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969.
13. J. Kloos, R. Majumdar, F. Niksic, and R. Piskac. Incremental, inductive coverability. In *CAV*, pages 158–173, 2013.
14. L. Lamport. The mutual exclusion problem: Part II—statement and solutions. *J. ACM*, 33(2):327–348, 1986.
15. R. Majumdar, R. Meyer, and Z. Wang. Static provenance verification for message passing programs. In *Proceedings of the 20th Static Analysis Symposium*, SAS'13. Springer-Verlag, 2013.
16. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
17. C. Rackoff. The covering and boundedness problems for vector addition systems. *Theor. Comput. Sci.*, 6:223–231, 1978.
18. W. Reisig. *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013.
19. A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons Ltd., 1986.
20. A. Solar-Lezama, L. Tancau, R. Bodík, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.
21. A. Valmari and H. Hansen. Old and new algorithms for minimal coverability sets. In *Petri Nets*, pages 208–227, 2012.
22. H. Wimmel and K. Wolf. Applying CEGAR to the Petri net state equation. *Logical Methods in Computer Science*, 8(3), 2012.