Invisible invariants in the spotlight

Verifiable proofs for parameterized systems

Mikhail Raskin[0000−0002−6660−5673], Christoph Welzel[0000−0001−5583−0640], and Javier Esparza[0000−0001−9862−4919]

Technical University Munich,
Boltzmannstraße 3,
85748 Garching bei München
{raskin,welzel,esparza}@in.tum.de

Abstract. We study the problem of automatically proving parameterized mutual exclusion algorithms mutually exclusive. In our first contribution we show that the problem remains undecidable even for a very weak model of computation. In this model agents have no identities but can iterate over all agents. This iteration, however, is executed in an arbitrary sequence which is only guaranteed to contain every agent at least once. Further, we show that checking if a set of configurations describable in first-order logic is an invariant is also undecidable.
In our second contribution we present an automatic procedure, based on first-order theorem proving, that constructs small and readable inductive invariants of a given algorithm. This procedure leverages invariants that can be used to prove finite instances of the parametric algorithms correct. Formulating generalizations of these invariants in first-order logic allows us to use the mature tooling of automated theorem proving to discharge required proof obligations. Moreover, we can give externally verifiable *certificates* of positive results as a sequence of first-order problems which collectively prove the desired property of the parameterized system. We show that this technique is able to automatically produce modular proofs of mutual exclusion for basic algorithms from the literature.

Keywords: parameterized verification · first-order theorem proving · inductive invariants · mutual exclusion

1 Introduction

Replicated systems consist of a fully symmetric finite-state program executed by an unknown number of indistinguishable agents, communicating by rendezvous or via shared variables with finite domain. They are a special class of *parameterized systems*, that is, infinite families of systems that admit a finite description in some suitable modeling language. In the case of replicated systems, the (only) parameter is the number of agents executing the program.

Verifying a replicated system amounts to proving that an infinite family of systems satisfies a given property. Despite this fact, some verification problems, in particular many safety problems, have been proved decidable for numerous

classes of systems using results from the theory of vector addition systems and the theory of well-quasi-orders [7]. The classes are able to model numerous interesting communication protocols and cache-coherence protocols. However, somewhat frustratingly, none of them can model the most prominent and famous set of benchmarks for verification of concurrent systems: Mutual exclusion algorithms for an arbitrary number of processes. Indeed, to the best of our knowledge all formal models of replicated systems proposed so far are either too weak to model mutual exclusion algorithms, or are Turing powerful. This result has prompted the invention by Abdulla *et al.* of several semi-decision procedures that have been able to successfully prove correctness of many properties for an arbitrary number of processes [2, 1, 6, 3], both for the simplified versions of the algorithms and for the more faithful ones without global guards.

In this paper we present two contributions. First, we ask ourselves if there can be a model of computation able to model mutual exclusion algorithms, and for which checking the mutual exclusion property is decidable. At first sight, this question could seem to have been answered by Emerson and Kahlon in [20]. They present a formal model of computation by indistinguishable processes with *global guards* in which a process can do an action if all other processes occupy certain states, and show that it is Turing powerful. Since mutual exclusion requires that a process checks the state of every other process, every formalism able to capture mutual exclusion algorithms is deemed to be Turing powerful, qed. However, this argument is invalid, because in [20] the process can check the state of all other processes *in a single atomic step*, a capability that greatly simplifies the design of mutual exclusion algorithms, and which none of the classical algorithms assumes. In the first part of the paper, we present a far weaker model, in which processes can loop over other processes, one at a time, with a termination guarantee but with no guarantee that processes are visited in any order. We show that, despite these constraints, deciding the mutual exclusion property is undecidable. Interestingly, the proof makes extensive use of the fact that the model captures mutual exclusion algorithms. This result provides strong evidence that for every model of replicated systems able to capture mutual exclusion algorithms, the parameterized mutual exclusion problem is undecidable.

Our second contribution addresses the question of producing explainable proofs of mutual exclusion. Existing semi-decision procedures based on regular model-checking, or abstraction techniques plus well quasi-orders, are based on symbolic state-space exploration; they automatically compute an overapproximation of the set of reachable configurations of the system, which can be interpreted as one single, monolithic invariant implying the mutual exclusion property [2, 1, 6, 3]. While these techniques have had remarkable success, the invariant is monolithic and difficult to understand by humans. For example [5] uses a set of 222 words of length 2 to represent an abstraction of reachable states for Dijkstra's algorithm for mutual exclusion. We propose an alternative approach. Previous work has shown that in many mutual exclusion algorithms and other systems, safety properties are implied by very simple *syntactic invariants* that, for each fixed number of processes, can be automatically computed from the

syntactic description of the algorithm, without state-space exploration [22, 21]. The invariants are of the form "this set of states is always occupied by at least one (exactly one) process". Recently, we have even shown that for certain classes of replicated systems the statement "for all instances, the syntactic invariants imply the mutual exclusion property" can be formalized as formula of monadic second-order logic on words and automatically checked, yielding automatic parameterized correctness proofs for several cache-coherence protocols and other systems. However, this automatic approach cannot be applied to mutual exclusion algorithms, because monadic second-order logic is not powerful enough in this case. For this reason, we present a semi-decision procedure that computes such invariants for instances of the algorithm with a small number of processes, and attempts to generate parameterized invariants valid for any number of processes. The parameterized invariants are captured by first-order formulae. A dedicated procedure generates invariant candidates, which are then checked using a first-order theorem prover.

Our technique bears resemblance, and is inspired by, the work on invisible invariants carried out by Pnueli *et al.* in the early 00's [29, 8]. However, we align our contribution more with recent observations regarding invisible invariants from [37]. Rather than trying to obtain a sufficient abstraction from a single instance of the parameterized system we inspect instances of increasing sizes and try to abstract inductive invariants of finite instances to inductive invariants of the parameterized system.

2 A very weak model of distributed programs

We introduce a program model to faithfully model classic mutual exclusion algorithms but which is as weak as possible w.r.t. the communication of agents.

Syntax. A program is a list $q_1 \colon \mathrm{comm}_1; \ldots; q_n \colon \mathrm{comm}_n$ of labelled commands over a set V of Boolean variables. All labels are distinct, and we call $Q = \{q_1, \ldots, q_n\}$ the set of *local states*. A program is executed by a finite but arbitrarily large set of indistinguishable agents, each of them with a local copy of all the variables. At each moment in time, each agent is in a local state, and one of the agents executes a command, while the others stay put. Commands are of the form

$$v := b \textbf{ goto } q \qquad \text{or} \quad \textbf{if foreach } v = b \textbf{ then goto } q \textbf{ else goto } q'$$

where $v \in V$, $b \in \{0, 1\}$, and $q, q' \in Q$. We call them *assignments* and *loop statements*, respectively.

Informal semantics. The semantics of assignments is standard; an agent A executes $v := b$ **goto** q by updating their local copy of v and moving to q. An agent executes an loop statement by checking whether the variable v of *all* agents has value b, and moving to state q or q' depending on the result. However, the agent does *not perform the check in an atomic step*. Instead, the agent, say A,

repeatedly checks whether the variable v of some other agent B has value b until A has inspected every other agent a least once. If $v_B = b$, then A continues its iteration (or if A has already inspected every other agent at least once moves to q); if $v_B \neq b$ then A moves to q'. The semantics only guarantees that if A only sees agents with $v = b$, then A will eventually pick all agents, and then A will finally move to q. This behaviour is formalized in the next paragraph.

Formal semantics. The semantics of a program P assigns to each *instance* $\langle P, N\rangle$ of P, in which $N \in \mathbb{N}$ agents execute P, a set of *runs*. Intuitively, $[N] = \{1, \ldots, N\}$ is the set of *process identities*. A run is an infinite sequence of configurations. In order to define Run_N we introduce configurations and schedules.

A *configuration* is a triple $C = \langle \mathrm{pc}, \mathrm{val}, \mathrm{S}\rangle$, where $\mathrm{pc}\colon [N] \to Q$ is the *state mapping* that assigns to each process its current local state; $\mathrm{val}\colon V \times [N] \to \{0, 1\}$ is the *variable mapping* that assigns to each variable v of each process $i \in [N]$ the current value $\mathrm{val}(v, i)$ of v in i; and $\mathrm{S}\colon [N] \to 2^{[N]}$ is the *seen mapping*. For every $i \in [N]$, if $\mathrm{pc}(i)$ is the label of an loop statement, then $\mathrm{S}(i)$ is the set of agents that process i has picked so far; if $\mathrm{pc}(i)$ is the label of an assignment, then $\mathrm{S}(i) = \emptyset$. Given $i \in [N]$, we define when the instance (P, N) can make an *i-step* from $C = \langle \mathrm{pc}, \mathrm{val}, \mathrm{S}\rangle$ to $C' = \langle \mathrm{pc}', \mathrm{val}', \mathrm{S}'\rangle$, denoted $C \xrightarrow{i} C'$. Let $\mathrm{pc}(i) = q$. There are two possible cases.

- The command labeled by q is $q : v := b'$ **goto** q'.
  Then $C \xrightarrow{i} C'$ if $\mathrm{pc}' = \mathrm{pc}[i \mapsto q']$, $\mathrm{val}' = \mathrm{val}[(v, i) \mapsto b']$, and $\mathrm{S}' = \mathrm{S}[i \mapsto \emptyset]$.
- The command labeled by q is $q\colon$ **if foreach** $v = b$ **then goto** q' **else goto** q''.
  Then $C \xrightarrow{i} C'$ if $\mathrm{val}' = \mathrm{val}$ and additionally
  - $\mathrm{S}(i) = [N] \setminus \{i\}, \mathrm{pc}' = \mathrm{pc}[i \mapsto q'], \mathrm{S}' = \mathrm{S}[i \mapsto \emptyset]$; or
  - there exists $j \in [N] \setminus \{i\}$ with $\mathrm{val}(v, j) \neq b, \mathrm{pc}' = \mathrm{pc}[i \mapsto q''], \mathrm{S}' = \mathrm{S}[i \mapsto \emptyset]$; or
  - there exists $j \in [N] \setminus \{i\}$ with $\mathrm{val}(v, j) = b, \mathrm{pc}' = \mathrm{pc}, \mathrm{S}' = \mathrm{S}[i \mapsto \mathrm{S}(i) \cup \{k\}]$,

  where $f[e \mapsto r](e) = r$ and $f[e \mapsto r](x) = f(x)$ for all $x \neq e$.

A *schedule* is a mapping $\sigma\colon \mathbb{N} \to [N]$ such that for every $i \in [N]$ there are infinitely many $t \in \mathbb{N}$ with $\sigma(t) = i$. Intuitively $\sigma(t) = i$ means that process i is the one that executes a statement at time t. A *σ-run* is an infinite sequence $\rho = C_0 C_1 C_2 \cdots$ of configurations such that

- C_0 is the configuration in which all processes are at their initial locations and all variables have value 0;
- $C_j \xrightarrow{\sigma(j)} C_{j+1}$ for every $j \geq 0$;
- every process makes infinitely many steps; and
- for every process $i \in [N]$ there are infinitely many C_j such that $\mathrm{S}_j(i) = \emptyset$. (This condition ensures that every loop statement terminates.)

A configuration is *reachable* if it is contained in some run.

Our language is powerful enough to implement the following mutual exclusion algorithm, akin to the one in [18].

Example 1 (Mutex algorithm). Consider the following program with one single variable b, where `noop` is an instruction that does nothing, with the same sematics as $b := b$:

```
  initial: b:=1 goto loop
     loop: if foreach b=0 then goto critical
                          else goto break
    break: b:=0 goto initial
 critical: noop goto done
     done: b:=0 goto initial
```

Proposition 1. *The program P of Example 1 satisfies the mutual exclusion property w.r.t. the local state* `critical`*, that is,* $|\{i \in [N] \mid \mathrm{pc}(i) = \texttt{critical}\}| \leq 1$ *holds for every reachable configuration of every instance $\langle P, N\rangle$ of P.*

Proof. The following statements are inductive in P, i.e., if any reachable configuration C of any instance $\langle N, P\rangle$ satisfies the statement and $C \to C'$, then C' also satisfies it:

- for all $i \in [N]$: $\mathrm{val}(b, i) = 1$ or $\mathrm{pc}(i) = \texttt{initial}$;
- for all $i \neq k \in [N]$ then either $\mathrm{val}(b, i) = 0$ or $\mathrm{val}(b, k) = 0$ or $k \notin \mathrm{S}(i)$ or $i \notin \mathrm{S}(k)$.

These statements allow to deduce the inductivity of the mutual exclusion property which gives the desired result.

3 Undecidability results

In this section we prove that even for the presented weak programming model the problem whether a given algorithm maintains the mutual exclusion property is undecidable. Moreover, we establish that checking hypothesis for inductive invariants in the natural logical structure is undecidable.

3.1 Mutual exclusion is undecidable

We prove that whether a program in our very weak programming language satisfies the mutual exclusion property is undecidable. Interestingly, the proof makes heavy use of the fact that the language allows one to implement a mutual exclusion algorithm.

For the proof we show that programs can simulate two counter machines. Initially, we choose a leader by running a mutual exclusion algorithms and electing the first agent to enter the critical section. Then, this leader runs the algorithm of the two counter machine with all other agents collectively storing the counter values, and using mutual exclusion to ensure correctness of operations.

Proposition 2 (Undecidable mutual exclusion). *Given a program P and a local state q, the question whether P satisfies the mutual exclusion property w.r.t. q is undecidable.*

Proof. To keep the presentation more concise we allow processes to check their current state; i.e., we introduce a command

$$\textbf{if } v_1 = b_1 \textbf{ then } v_2 := b_2 \textbf{ goto } q_1 \textbf{ else goto } q_2$$

with the obvious semantics that a process checks its current v_1 value and if it coincides with b_1 executes the assignment $v_2 := b_1$ and otherwise moves on to label q_2. Note that this does not increase the expressive power of the programming model since these checks can be statically resolved by introducing a separate label for every of the finitely many possible valuation of local variables.

We simulate a two counter machine. For this, we initially elect a leader which will simulate the behaviour of the machine while all other agents collectively represent the values of the counters. We will present how to simulate an increment and a decrement of both counters and, additionally, how to check if any counter currently holds value 0. The "zero checks" are easily carried out by simple iterations of the leader over all other agents. The increment and decrement operations are a little bit more involved. Generally speaking, these operations follow a *call-response* pattern; i.e., the leader issues a command and all non-leader agents react to this command. All these simulations crucially depend on the fact that we can enforce that certain labels are mutually exclusive. This is essential not only for the initial leader election but also to execute increments or decrements only once. For this all non-leader agents compete in a mutual exclusion algorithm to determine the order in which they may attempt to execute the currently issued command. In the following, we will present the details of this overview.

We start by running the mutual exclusion algorithm presented in Example 1 to elect a leader. That is, we execute

```
initial: b:=1 goto loop
loop: if foreach b=0 then goto critical
      else goto break
break: b:=0 goto initial
critical: if foreach leader=0 then goto grabLeader
          else goto becomeFollower
grabLeader: leader:=1 goto ℓ
becomeFollower: follower:=1 goto f
```

This gives us two labels ℓ and f which a process moves to after this initial step. It is crucial to observe that exactly one process moves to the label ℓ while all *other* processes move to f. Both labels start with a `b:=0` command which releases the critical section and allows the other processes to proceed. From that point onwards all follower agents execute the same steps in an infinite loop. Namely, they

1. scan if there is an issued command,
2. compete in a mutual exclusion algorithm to execute the command,
3. try to execute the command,
4. signal success or failure,
5. and wait for the leader to globally synchronize and reset all followers.

To this end, every follower repeatedly checks if the leader issued an command and tries to execute it. The leader indicates that currently a command is executed

by setting its commandIssued value. The followers constantly check if *any* agent set this value (which only ever is the leader since followers do not set this value ever). This is easily achieved using a foreach command:

```
waitForCommand: if foreach commandIssued=0 then goto waitForCommand
                else goto executeCommand
```

Once an agent with the commandIssued bit is observed all followers compete in a mutual exclusion algorithm to determine in which order they try to execute the command.

```
executeCommand: f:=1 goto loop
loopCommand: if foreach f=0 then goto execute
             else goto breakCommand
breakCommand: f:=0 goto executeCommand
execute: <executing-command-section>
```

This is again just a copy of the algorithm used in Example 1. In <executing-command-section> the process checks which command was issued by inspecting which bit the leader actually set. Hence, this section looks as follows

```
execute: if foreach successCommand=0 goto checkInc1
         else goto failCommand

checkInc1: if foreach cmdInc1=0 then goto checkInc2
           else goto inc1
checkInc2: if foreach cmdInc2=0 then goto checkDec1
           else goto inc2
checkDec1: if foreach cmdDec1=0 then goto checkDec2
           else goto dec1
checkDec2: if foreach cmdDec2=0 then goto checkHalt
           else goto dec2
checkHalt: if foreach cmdHalt=0 then goto checkInc1
           else goto halt
inc1: if value1=0 then value1 := 1 goto setSuccessCommand
      else goto failCommand
inc2: if value2=0 then value2 := 1 goto setSuccessCommand
      else goto failCommand
dec1: if value1=1 then value1 := 0 goto setSuccessCommand
      else goto failCommand
dec2: if value2=1 then value2 := 0 goto setSuccessCommand
      else goto failCommand
halt: f:=0 goto halt
```

Moving to the label setSuccessCommand makes the process set its f value again to 0 to release the critical section, set its successCommand value to 1, and also set its endCommand value to 1. Moving to the label failCommand only sets the f value to 0, and the endCommand value to 1. Eventually, both execution paths move to the following section:

```
waitForReset: if foreach reset=0 then goto waitForReset
              else goto reset1
reset1: successCommand := 0 goto reset2
reset2: endCommand := 0 goto waitForCommand
```

This section simply implements a mechanism for synchronization. That is, all followers await the signal of the leader to end the execution of the current command and start a new one.

On the other hand, the leader, initially, waits for every other process to become a follower by

```
waitForFollowers: if foreach follower=1 then goto command1
                  else goto waitForFollowers
```

where `command1` is the first command of the considered two counter machine (in the encoding that follows). This means the leader waits for all other processes to set their `follower` bit making sure every process passed the first step.

The leader can now issue one of the following five commands:

- increasing counter one by setting its `cmdInc1` and `commandIssued` bits,
- increasing counter two by setting its `cmdInc2` and `commandIssued` bits,
- decreasing counter one by setting its `cmdDec1` and `commandIssued` bits,
- decreasing counter two by setting its `cmdDec2` and `commandIssued` bits, and
- halting which moves all followers into the dedicated `halt` label.

Moreover, the leader can carry out checks for the value 0 in either counter by executing

```
zeroCheck1: if foreach value1=0 goto successfulZeroCheck1
            else goto failedZeroCheck1
```

or

```
zeroCheck2: if foreach value2=0 goto successfulZeroCheck2
            else goto failedZeroCheck2
```

respectively.

After issuing a command the leader needs to synchronize all followers again. For this the leader waits until all followers signal the completion of the command, issues the reset, and waits for the resets to be executed; i.e.,

```
waitForCompletion: if foreach endCommand=1 goto checkSuccess
                   else goto waitForCompletion
checkSuccess: if foreach successCommand=0 goto failedExecution
              else goto issueReset
issueReset: reset := 1 goto waitForReady
waitForReady: if foreach endCommand=0 goto unissueReset
              else goto waitForReady
unissueReset: reset := 0 goto nextCommand
```

Here the label `failedExecution` indicates that no process was able to execute the issued command. This can be the case if a counter is decremented beyond 0 or incremented over the size of the current instance. The leader can then just choose to idle indefinitly.

If the leader eventually reaches a halting state it can indicate so by issuing the `halt` command. Then every follower seeing this command being issued moves into the dedicated `halt` label. This renders this program mutually exclusive w.r.t. the label `halt` if and only if the simulated two counter machine does not halt. The undecidability of the problem follows from the undecidability of the halting problem for two counter machines.

3.2 Inductive invariance is undecidable

Mutual exclusion can be proved by guessing a candidate for an inductive invariant of the program, and then checking that the candidate is indeed inductive, and that it implies the mutual exclusion property. In this section we show that, unfortunately, checking inductiveness of sets of configurations expressible in a natural logical language is undecidable.

For the rest of the section we fix a program P with sets C and B of control states and variables.

Inductive invariants. For every $N \in \mathbb{N}$ let $\mathcal{C}_N$ and $\mathcal{C}_{0N}$ denote the sets of configurations and initial configurations of the instance (P, N) of P. The sets of configurations and initial configurations of P are $\mathcal{C} = \bigcup_{N\in\mathbb{N}} \mathcal{C}_N$ and $\mathcal{C}_0 = \bigcup_{N\in\mathbb{N}} \mathcal{C}_{0N}$. An inductive invariant of P is a set $I \subseteq \mathcal{C}$ such that $C \in \mathcal{C}$ and $C \to C'$ implies $C' \in \mathcal{C}$. An inductive invariant I implies the mutual exclusion property if $\mathcal{C}_0 \subseteq \mathit{Mutex}$, where Mutex is the set of all configurations of P with at most one process in the critical section.

Expressing sets of configurations Inductive invariants that imply the mutual exclusion property are infinite sets of configurations. In order to check the inductiveness of a candidate we first need a formal language for describing the candidate itself. The natural candidate is first-order logic over a signature corresponding to the structure of configurations. Given two sets C and B of local states and variables, let $\mathrm{FOL}(\mathrm{C}, \mathrm{B}, \mathrm{S})$ be the set of first-order formulas over the signature containing a unary predicate symbol for each element of $\mathrm{C} \cup \mathrm{B}$, and a single binary predicate symbol S. A configuration $C = \langle \mathrm{pc}, \mathrm{val}, \mathrm{S}\rangle$ induces a structure $\mathcal{C}$ over this signature. The universe of $\mathcal{C}$ is $[N]$. Further, $q^{\mathcal{C}}$ contains the processes that are at state q in C; $b^{\mathcal{C}}$ contains the processes for which variable b is true in C; and $\mathrm{S}^{\mathcal{C}}$ contains the pairs $\langle i, j\rangle$ of processes such that $j \in \mathrm{S}(i)$. From now on we identify the configuration C and the structure $\mathcal{C}$. Further, given a sentence φ of $\mathrm{FOL}(\mathrm{C}, \mathrm{B}, \mathrm{S})$, we let $[[\varphi]]$ denote the set of configurations that satisfy φ.

Observe that the converse does not hold, not every model of a sentence φ of $\mathrm{FOL}(\mathrm{C}, \mathrm{B}, \mathrm{S})$ induces a configuration. Only the models in which every process occupies exactly one local state do.

Undecidability of inductiveness. We first prove that the problem whether a formula of $\mathrm{FOL}(\mathrm{C}, \mathrm{B}, \mathrm{S})$ is satisfied by some configuration is undecidable.

Proposition 3. *The following problem is undecidable: Given a sentence φ of FOL$(\mathrm{C}, \mathrm{B}, \mathrm{S})$, does $[[\varphi]] = \emptyset$ hold? Moroever, the problem remains undecidable if φ belongs to FOL(B, S), i.e., does not mention any local state.*

Proof. Let φ be a formula of $\mathrm{FOL}(\mathrm{B}, \mathrm{S})$, and let P be any program with one single control state and B as set of variables. There is a bijection between the models of φ and the configurations of P. So $[[\varphi]] = \emptyset$ holds iff φ is satisfiable. By definition, $\mathrm{FOL}(\mathrm{B}, \mathrm{S})$ is the logic of one binary relation and arbitrarily many

unary predicates. By [24, Theorem 4] this is a conservative reduction class, and so the satisfiability problem is undecidable.

Using this proposition we can show:

Corollary 1. *The following problem is undecidable: Given a program P and a sentence φ of FOL*$(\mathrm{C}, \mathrm{B}, \mathrm{S})$*, is $[[\varphi]]$ an inductive invariant?*

Proof. By reduction from the problem whether $[[\varphi]] = \emptyset$ holds for a given formula φ of FOL(B, S). Given such a formula, let P be the result of appending to any program over B the following two commands:

```
loop: if foreach b=0 then goto loop else goto halt
halt: noop
```

where $b \notin \mathrm{B}$. Let $\varphi' = \varphi \wedge \forall x \ (b(x) \wedge \mathrm{loop}(x))$. We claim that $[[\varphi']]$ is an inductive invariant of P iff φ is unsatisfiable. If φ is unsatisfiable, then so is φ'. So $[[\varphi']] = \emptyset$, which is trivially inductive. If φ is satisfiable, then, since it is a formula of FOL(B, S) and $b \notin \mathrm{B}$, $[[\varphi']]$ contains at least one configuration C of P where all processes are in state `loop` and satisfy `b=0`. By the definition of P, we have $C \to C'$ for a configuration C' where at least one process satisfies is in `halt`. So $C' \notin [[\varphi']]$, and so $[[\varphi']]$ is not inductive.

4 Automatically proving mutual exclusion using parameterized invariants

Previous work co-authored by one of us has developed an invariant-based approach to the verification of safety properties of finite-state distributed systems with a fixed number of processes [22, 21]. (A similar approach is also used in [10].) The invariants are extracted from the syntactic representation of the system, without the need for state space exploration. The approach iteratively computes a configuration C, reachable or not, that satisfies the invariants computed so far but still violates the property. If no such C exists, the property is proved. Otherwise, an SMT-solver efficiently computes a new invariant that is not satisfied by C, thus proving C unreachable from the initial configuration.

This technique has been extended to certain restricted classes of parameterized systems in [14, 13]. In those systems, the problem whether the set of all invariants for all instances satisfies the mutual exclusion property can be reduced to the satisfiability problem for monadic second order logic on words. However, even though our programming language is very weak, programs written in it do not fit within these classes. This is because non-atomically checked global conditions require individual bookkeeping of the agents about which participants they already have inspected. This is modelled in [29, 8] as "unstratified" type structures, in [6] by a graph based representation and in [5] by a binary relation which is incorporated into views of the configurations. Drawing inspiration from [37] we present in the following an approach which is mostly aligned with invisible invariants. In contrast to the specifically tailored abstractions of graphs and

views from [6] and [5] respectively, we formulate the behaviour of our system as a first-order theory and our inductive invariants as formulae within this theory. Hence, we obtain a *certificate* for our proofs; namely, a sequence of first-order formulae which formulate every necessary proof obligation. These formulae are concise; even to the point that a user can read and comprehend the concepts expressed by the formulae. Moreover, every proof can be externally and independently verified by the constantly improving tools of automated theorem proving.

Since we have to give up decidability of inductiveness for invariants, we can consider the standard language used to describe mutual exclusion algorithms, more powerful than our very weak one. In the standard language agents have identities, which we assume to be the numbers $1, \ldots, N$, and agents use loop variables to loop over all agents in the order given by their identities.

Let P be such a program, and let $\langle P, N \rangle$ denote the program instance with N agents. We prooceed as follows.

(1) Starting with the smallest meaningful value for N (usually 2), we compute a set I of inductive invariants proving the property for the finite-state instance $\langle P, N \rangle$ using the approach of [21].
(2) We use an abduction technique to generalize I to a set of candidates for parameterized inductive invariants. A candidate is a formula of first-order logic that can be interpreted over configurations of any instance of P.
(3) We use a first-order theorem prover to confirm that a given candidate φ is inductive. For this we construct a first-order formula stating that if a configuration C of some instance satisfies φ and C' is its successor configuration, then C' also satisfies φ.
(4) We check, again with the help of a first-order theorem prover, if the parameterized invariants computed so far imply the property. If so, we are done. If not, we increase N and go to step (1).

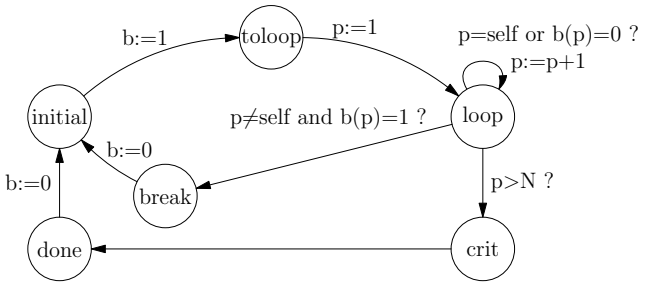

Fig. 1. Natural formalisation of Example 1 in the standard computation model with agent identities and ordered iteration.

In the rest of the section we describe the steps (1)-(4) in a semiformal but hopefully precise way. We use as running example the mutual exclusion algorithm

for N agents from Example 1, but with the standard loop semantics in which an agent loops through all agents in order $1, \ldots, N$; that is, we assume a linear topology for the parameterized system. For the presentation it is convenient to represent the program as an automaton with transitions operating on variables. The automaton is shown in Figure 1. Variable p is a loop iterator ranging over $\{0, 1, \ldots, N, N+1\}$, and `self` $\in \{1, \ldots, N\}$ is the identity of the agent. This means there are three variables for every agent: b, p, and pc. The corresponding specification for our tool can be seen in Figure 2.

```
local b  [false, true]
local pc [initial, toloop, loop, break, crit, done]
local p  [0 - succ(N)]
```

Fig. 2. Specification of variables for Example 1 from Figure 1; here `succ` describes the function $(\cdot) + 1$.

For example, agent p_1 can execute the transition going from state `loop` to state `loop` if either the current value of p is 0 or p_1, or $p(p_1)$ points to an agent and the variable b of the agent $p(p_1)$ has value `false`; in any case taking the transition increases the value of p by 1. We let $\mathbf{D}$ denote the complete program of Figure 1.[1]

4.1 Computing disjunctive invariants of specific instances

Once we fix a value for N we obtain for the variable p the possible values $P = \{0, \ldots, N, N+1\}$. Regardless of the value of N the variables pc and b have finite sets of values; namely, $Q = \{\texttt{initial}, \texttt{toloop}, \texttt{loop}, \texttt{break}, \texttt{crit}, \texttt{done}\}$, and $B = \{\texttt{true}, \texttt{false}\}$ respectively. Consequently, we can capture the state of an instance using propositional variables

$$\begin{aligned} S_N = &\{\texttt{pc(j)=q} \mid \texttt{j} \in \{1, \ldots, N\}, \texttt{q} \in Q\} \\ &\cup \{\texttt{b(j)=v} \mid \texttt{j} \in \{1, \ldots, N\}, \texttt{v} \in B\} \\ &\cup \{\texttt{p(j)=k} \mid \texttt{j} \in \{1, \ldots, N\}, \texttt{k} \in P\} \end{aligned}$$

with the obvious intended meanings: "agent j is in state q", "variable b of agent j is true (false)", and "loop variable p of agent j has value k".

An inductive invariant of the instance $\langle \mathbf{D}, N \rangle$ is a propositional formula φ over S_N such that for every two configurations C, C' of $\langle \mathbf{D}, N \rangle$, if $C \models \varphi$ and $C \to C'$, then $C' \models \varphi$. Let us see how to use the technique of [21], to compute the following two inductive invariants for the instance $(\mathbf{D}, 3)$ directly from Figure 1, without exploring the reachable configurations of $(\mathbf{D}, 3)$:

$$(I_1) : \texttt{b(1)}{=}\text{true} \vee \texttt{pc(1)=initial}$$

[1] The specification of $\mathbf{D}$ can be found as the example `dijkstra-cutdown.her` in the publication of our tool `heron` [36].

$$(I_2): \begin{pmatrix} \texttt{pc(3)=toloop} \vee \texttt{b(3)=false} \vee \texttt{p(3)=1} \\ \vee \texttt{pc(1)=toloop} \vee \texttt{b(1)=false} \vee \texttt{p(1)=1} \\ \vee \texttt{p(1)=2} \vee \texttt{p(1)=3} \end{pmatrix}$$

Intuitively, the technique searches for subsets $R \subseteq S_3$ such that every transition of $\langle \mathbf{D}, 3 \rangle$ that sets some variable of R to false also sets at least one variable of R to true. Such subsets are called *traps* in [21], and can be easily computed using a SAT-solver. If R is a trap, then the disjunction of the variables of R is an inductive invariant. For example, $R = \{\texttt{b(1)=true}, \texttt{pc(1)=initial}\}$ is a trap, because a transition sets `b(1)` to `false` iff it also sets `pc(1)=initial` to true. This trap leads to invariant (I_1). On the other hand, (I_2) is the manifestation of the second statement in the proof of Proposition 1 (adapted to the linear topology). Its inductiveness is mainly rooted in the fact that process 1 can move its p-pointer past the value 3 only if process 3 has not set its b-flag. The same applies for process 3 w.r.t. process 1. Both processes set their b-flag whilst moving into the `toloop` state where both process set the p-pointer to 1. This establishes the inductiveness of (I_2).

4.2 Generating candidates

We present a heuristic abduction technique to guess candidates for parameterized inductive invariants (PINs) of $\mathbf{D}$ from inductive invariants of the instances $\langle \mathbf{D}, N \rangle$ for $N = 1, 2, 3, \ldots$. (While illustrated for $\mathbf{D}$, the general technique can be applied to any program. In fact, all following examples are adapted manifestations from actual executions of our implementation.)

Let us briefly discuss which logical language we use to formulate PIN-candidates. Mainly, we aim for a logical theory as concise as possible while maintaining enough expressiveness for the required program semantics. This semantics resolves around the linear topology of our parameterized systems and the consecutive iteration over this topology. Therefore, we choose the first-order theory of a totally ordered set with a minimal element: we fix the signature $\sigma = \langle 0, N, \mathrm{succ}, \leq, b, p, \mathrm{pc} \rangle$ and introduce a set of axioms $\mathcal{T}$ which states that $\leq$ is a total linear order with minimal element 0, succ is the unique successor (w.r.t. $\leq$) for every element, N is an element strictly larger than 0, and b, p, pc are function symbols. In the following, we will use constant symbols $1, 2, \ldots$ to refer to the terms $\mathrm{succ}(0), \mathrm{succ}(\mathrm{succ}(0)), \ldots$. For p we add $\forall x \,.\, 0 \leq p(x) \wedge p(x) \leq \mathrm{succ}(N)$ to $\mathcal{T}$ while for the enumeration types of b and pc we simply treat

- `false` as 0 and `true` as 1, and
- `initial` as 0, `toloop` as 1, `loop` as 2, and so on.

This can easily be encoded as axioms in $\mathcal{T}$.

We introduce the notion $p(x) \in [t_1, t_2]$ as an abbreviation for $t_1 \leq p(x) \wedge p(c) \leq t_2$. Formulae over this signature are interpreted on configurations in the expected way, where $C \models \varphi$ denotes that configuration C satisfies φ. For example, if

$$\varphi = \forall x \,.\, (1 \leq x \wedge x \leq N) \rightarrow p(x) \in [x, N]$$

and $C = \langle \mathrm{pc}, \mathrm{val}, p \rangle$ is a configuration of $\langle \mathbf{D}, 3 \rangle$, then $C \models \varphi$ iff $p(1) \in [1,3]$, $p(2) \in [2,3]$, and $p(3) \in [3,3]$. A closed formula φ is a PIN of a program if for every two configurations C, C' of any instance, if $C \models \varphi$ and $C \to C'$, then $C' \models \varphi$.

We present a heuristic abduction technique to guess PIN-candidates of the form

$$\forall x_1 \ldots \forall x_n \;.\; \mathrm{diff}(x_1, \ldots, x_n) \to \mathrm{guess}(x_1, \ldots, x_n)$$

where diff expresses that all of $x_1, \ldots, x_n$ are different, and $\mathrm{guess}(x_1, \ldots, x_n)$ is a quantifier-free formula, which (hopefully) captures inductive invariants of concrete instances. W.l.o.g. we can assume that the invariants for concrete instances obtained from traps (like the invariants I_1 and I_2 of $\langle \mathbf{D}, 3 \rangle$) are in the following normal form:

- Each disjunct is of the form

$$\texttt{pc(i)=q},\ \texttt{p(i)} \in [t_1, t_2], \text{ or } \texttt{b(i)=v}$$

  for $i \in [1, N]$, $t_1, t_2 \in P$, $q \in Q$, and $v \in B$.
- If $\texttt{p(i)} \in [t_1, t_2]$ and $\texttt{p(i)} \in [t_3, t_4]$ are both disjuncts, then the intervals $[t_1, t_2]$ and $[t_3, t_4]$ do not overlap and are not adjacent; i.e., their union is not a larger interval.

Note that this gives for I_2 the equivalent formula

$$\begin{aligned} &\texttt{pc(3)=toloop} \vee \texttt{b(3)=false} \vee \texttt{p(3)} \in [1,1] \\ \vee &\texttt{pc(1)=toloop} \vee \texttt{b(1)=false} \vee \texttt{p(1)} \in [1,3]. \end{aligned}$$

Initially, we check all occuring variables and collect the process indices these belong to. For the invariant I_2 this gives the process indices $\{1, 3\}$ since $p(1)$, $b(1)$ and, respectively, $p(3)$, $b(3)$ are part of I_2. Similarly to [29, 8] we then substitute the actual occurences of 1 and 3 with variables x_1 and x_2. This substitution as well as the value of N is logged in an interpretation $\{x_1 \mapsto 1, x_2 \mapsto 3, N \mapsto 3\}$ which we call a *context*. Actually, we include in the current context also constants for all terms restricting a domain; e.g. context additionally contains the interpretation of a constant symbol $c_0 \mapsto 4$ which corresponds to the term $\mathrm{succ}(N)$ in the domain restriction of p. Now, we formulate for $\mathrm{guess}(x_1, x_2)$ formulae which – when instantiated with this context – yield again I_2:

- We replace the array arguments 1 and 3 by variables x_1 and x_2 respectively, which gives us

$$\begin{aligned} I_2' = &\texttt{pc(}x_2\texttt{)=toloop} \vee \texttt{b(}x_2\texttt{)=false} \vee \texttt{p(}x_2\texttt{)} \in [1,1] \\ \vee &\texttt{pc(}x_1\texttt{)=toloop} \vee \texttt{b(}x_1\texttt{)=false} \vee \texttt{p(}x_1\texttt{)} \in [1,3]. \end{aligned}$$

- Then, we generate candidates for the intervals $[1,3]$ and $[1,1]$. Since $\mathrm{context}(N) = 3$ we generate, among others, the candidates $[x_1, x_2]$, and $[x_1, N]$ for $[1,3]$. Conceptually, we try for any $[c_1, c_2]$ elements in

$$\left\{ [t_1, t_2] \colon \langle t_1, t_2 \rangle \in \mathrm{context}^{-1}(c_1) \cup \{c_1\} \times \mathrm{context}^{-1}(c_2) \cup \{c_2\} \right\},$$

and also we try open intervals which lead to the instance $[c_1, c_2]$; e.g., t_1 and t_2 from the current context such that $[c_1, c_2]$ conincides with $(t_1, t_2]$, $[t_1, t_2)$, or (t_1, t_2).

The following PIN-candidates are, among others, examples which are generated for I_2' in context $\{x_1 \mapsto 1, x_2 \mapsto 3, N \mapsto 3\}$:

$$\begin{aligned} \forall x_1 \forall x_2 \,.\, x_1 \neq x_2 \rightarrow \quad & \mathrm{pc}(x_1) = \texttt{toloop} \vee b(x_1) = \texttt{false} \\ & \vee \mathrm{pc}(x_2) = \texttt{toloop} \vee b(x_2) = \texttt{false} \\ & \vee p(x_1) \in [x_1, x_2] \vee p(x_2) \in [x_1, x_1] \end{aligned} \tag{1}$$

$$\begin{aligned} \forall x_1 \forall x_2 \,.\, x_1 \neq x_2 \rightarrow \quad & \mathrm{pc}(x_1) = \texttt{toloop} \vee b(x_1) = \texttt{false} \\ & \vee \mathrm{pc}(x_2) = \texttt{toloop} \vee b(x_2) = \texttt{false} \\ & \vee p(x_1) \in [1, x_2] \vee p(x_2) \in [1, x_1] \end{aligned} \tag{2}$$

We can exclude some of these candidates by checking if they yield inductive invariants for all possible values of x_1 and x_2. For example, the context $\{x_1 \mapsto 2, x_2 \mapsto 3, N \mapsto 3\}$ gives for candidate (1)

$$\begin{aligned} & \texttt{pc(2)=toloop} \vee \texttt{p(2)} \in [2, 3] \vee \texttt{b(2)=false} \\ \vee & \texttt{pc(3)=toloop} \vee \texttt{p(3)} \in [2, 2] \vee \texttt{b(3)=false}. \end{aligned} \tag{3}$$

This, however, is not an inductive invariant of $\langle \mathbf{D}, 3 \rangle$ because a valid execution can put the second and third process both into the state `loop` and their respective p variables to value 1. Since this configuration is not captured by (3) it cannot be an inductive invariant.

Also, we refine our procedure for the formula $\mathrm{diff}(x_1, \ldots, x_n)$ by trying PINs which order x_1 to x_n linearly. Additionally, if this is consistent with the observed trap $\mathrm{diff}(x_1, \ldots, x_n)$ can also be used to enforce that $x_1 = 1$ or $x_n = N$ (or both).

We now proceed to the question how to use a first-order theorem prover to check that a candidate is a PIN.

4.3 Proving that a candidate is a PIN

We embed the execution steps of our programs in first-order logic. For this we introduce a second primed copy of every function symbols. That is, we consider the signature

$$\tau = \langle 0, N, \mathrm{succ}, \leq, b, p, \mathrm{pc}, b', p', \mathrm{pc}' \rangle \,.$$

Transitions can now easily be axiomatized as τ-formulae which relate a state with its successor state. That is, values of the primed function symbols are related to the values of the unprimed function symbols to express the changes. For example, any transition of state `loop` which advances p then contains the axiom $p'(p_1) = \mathrm{succ}(p(p_1))$. Now, a PIN-candidate φ is a PIN if

$$\{\mathcal{T}, \varphi, \mathrm{Step}\} \models \varphi' \tag{4}$$

where φ' is obtained from φ by replacing every occuring function symbol of a variable by its primed copy and Step is a disjunction of all possible transition formulae.

4.4 Proving mutual exclusion

Attempting to prove mutual exclusion is essentially very similar to Formula (4). Basically, we want to prove the inductiveness of the formula

$$\psi = \forall x_1 \forall x_2 \ . \ \mathrm{pc}(x_1) = c \wedge \mathrm{pc}(x_2) = c \rightarrow x_1 = x_2$$

where c is the state which is considered "critical". However, for this inductiveness check we add the set of all identified inductive invariants to our "knowledge base". That is, if I is the set of all identified PINs we dispatch

$$\{\mathcal{T}, \psi, \mathrm{Step}\} \cup I \cup \{\varphi' \colon \varphi \in I\} \models \psi' \tag{5}$$

to a theorem prover.

5 Results

We built a prototype, called `heron`, to test our approach on well-known examples from the literature. This prototype and all examples are available at [36]. We consider the mutual exclusion algorithms of

- Dijkstra [18],
- Example 1,
- de Bruijn [16],
- Eisenberg and McGuire [19],
- Knuth [25],
- Szymanski [33], and
- Burns [27].
- Taubenfeld, which is a variant of the bakery algorithm with bounded values for every variable [34],

We extend the logical representation with constant symbols to model global variables. We find sufficiently strong PINs to prove mutual exclusion for all examples for which the method of [21] proves the instantiations. This is possible for all examples but Taubenfeld's variant of the bakery algorithm, and the algorithms of Szymanski and Burns. For these examples the method of [21] cannot prove the mutual exclusion property even for only *two* agents. We present the results of our tool in the table of Figure 3. All experiments were carried out on a 8th generation i7 processor on a single core which – during running the experiments – operated with a clock speed between 3.7 and 3.8 GHz.

5.1 Evaluation

In comparison to [5] we fail to prove the algorithms of Burns as well as Szymanski. There are no data for the algorithms of Knuth, Eisenberg-McGuire and de Bruijn in [5]. For a direct comparison we use the non-atomic version of Dijkstra's

Example	Result	Time (in s)	max. N	# traps	# used invariants
Dijkstra	Success	82 (041, 124)	3.4 (3, 4)	12.6 (11, 14)	4.1 (4, 5)
Knuth	Success	111 (070, 243)	4.2 (4, 6)	21.5 (17, 31)	6.0 (5, 7)
Eisenberg-McGuire	Success	081 (080, 083)	4.0 (4, 4)	12.3 (11, 14)	5.2 (4, 6)
de Bruijn	Success	270 (211, 562)	4.0 (4, 4)	18.1 (16, 24)	6.0 (5, 8)
Example 1	Success	048 (033, 065)	3.5 (3, 4)	09.6 (08, 11)	3.0 (3, 3)
Bakery	Failed (finite instance)	-	2	-	-
Szymanski	Failed (finite instance)	-	2	-	-
Burns	Failed (finite instance)	-	2	-	-

Fig. 3. The results of `heron`. Values are the mean of ten computations. The minimum and maximum value are given in brackets. The second column states the result, and the following columns report the required time (as given by the `time` command), the maximum value of N for which the system was instantiated and the number of found traps across all instantiations, and the number of invariants required to establish the mutual exclusion property. We used `Vampire` to compute proofs for all inductiveness checks. The values for "used invariants" are determined by the number of distinct occuring in these proofs.

algorithm. Here [5] reports success with 222 views. Since "the set of views collectively represent a set of reachable configurations" [5] we argue that the four or five invariants formulated in first-order logic, which are necessary to establish the inductiveness of mutual exclusion, are easier to understand by a user than a set of 222 views. This is where we see the crucial distinction between view abstraction and `heron`. Essentially, the set V of 222 views of size 2 describes *one* formula of the form $\forall x_1 \forall x_2 \,.\, x_1 < x_2 \rightarrow \bigvee_{view \in V} view(x_1, x_2)$ where *view* is a formula describing the current state of the processes at position x_1 and x_2 (and potentially their relative state to each other regarding iteration variables). On the other hand, `heron` computes multiple individual invariants of the form $\forall x_1 \ldots \forall x_n \,.\, \varphi$ where φ is a disjunction of simple atoms. The abstraction `heron` computes is then the conjunction of all these individual invariants. Since the invariants are of a simple structure they can be individually examined and understood. Noteworthingly, we observe that for all examples `heron` computes invariants $\forall x_1 \ldots \forall x_n \,.\, \varphi$ where $n \leq 2$. The comparison with [6] follows the same pattern. Here Dijkstra's algorithm is proven mutually exclusive by computing 41 constraints in form of upward-closed sets of graphs. We re-iterate that we believe our first-order formulae are a more readable abstraction. To illustrate this point we show a set of invariants for our running example in Figure 4 which suffices to prove the mutual exclusion property. However, it is important to observe that we pay a hefty price w.r.t. computation time for the readability of our invariants; namely, at least two orders of magnitude in comparison with [5, 6].

In contrast to previous work around invisible invariants [8, 29] `heron` iteratively inspect larger and larger instances of the parameterized system to gradually refine its abstraction. Again, `heron` needs a small set of invariants with at most two quantified processes to establish the mutual exclusion property for the examples above while in [8] the examined instances have at least four processes and, con-

$$\forall x_1 \,.\, b(x_1) = \texttt{true} \vee \mathrm{pc}(x_1) = \texttt{initial} \tag{6}$$

$$\forall x_1 \,.\, \begin{pmatrix} \mathrm{pc}(x_1) = \texttt{break} \vee \mathrm{pc}(x_1) = \texttt{loop} \\ \vee \mathrm{pc}(x_1) = \texttt{toloop} \vee \mathrm{pc}(x_1) = \texttt{initial} \vee p(x_1) = \mathrm{succ}(N) \end{pmatrix} \tag{7}$$

$$\forall x_1 \forall x_2 \,.\, \left(x_1 \neq x_2 \rightarrow \begin{bmatrix} b(x_1) = \texttt{false} \vee \mathrm{pc}(x_1) = \texttt{toloop} \vee p(x_1) \in [1, x_2] \\ \vee b(x_2) = \texttt{false} \vee \mathrm{pc}(x_2) = \texttt{toloop} \vee p(x_2) \in [1, x_1] \end{bmatrix} \right) \tag{8}$$

Fig. 4. The invariants occuring in the proof for the the inductiveness of the mutual exclusion property of Example 1 by Vampire.

sequently, the synthesized inductive invariants are monolithic formulae with (at least) four universally quantified variables.

In [35] also universally quantified invariants are considered. However, the considered programming model and its logical embedding is based on reasoning about cardinalities of sets. Hence, the considered mutual exclusion algorithms are based on ticketing which is out of scope for our contribution but also does not allow for modelling of the iteration structure as honestly as we do.

5.2 Implementation details

In the following we discuss shortly some decisions we made regarding the actual implementation of the steps which we lay out in Section 4. In general, heron is implemented in the Python programming language. It implements the proving procedure for instances from [21] using Z3 [17] for finding integer solutions for linear equations and clingo [23] for SAT-solving.

In order to employ automated theorem proving for the Formula (4) and Formula (5) we render both in the commonly used TPTP format [32]. Since TPTP is a generally used format to represent first-order formulae it is supported by most automated theorem provers (cp. [9, 26, 31]). Moreover, since the formula Step is a disjunctive formula (where every disjunct models one transition), we actually perform these checks by individual calls to a theorem prover for every transition separately. This also helps to keep the problems which are dispatched to the used theorem provers concise. Separating these checks for Formula (5) allows us to leverage the increasing precision we get with more and more invariants; namely, as soon as we can establish the inductiveness of mutual exclusion with a set of PINs we do not need to check this transition again. Secondly, heron provides for all inductiveness checks a set of TPTP files: one to witness that the desired property is initially true and one to witness the inductiveness for every possible transition. We refer to the collection of these files as *certificate* of the proof. That is, heron does not only prove the mutual exclusion property but provides a *readable* and *verifiable* chain of reasoning – a certificate – for it.

heron relies mainly on CVC4 [9] as theorem prover. To approach the inherent undecidability of the checks for Formula (4) and Formula (5) we set timeouts for these checks. If these timeouts are exceeded we consider the check as failed.

Empirical knowledge led to timeouts of 1s for Formula (4) and 30s timeouts for Formula (5) (both per transition). Although CVC4 proves very powerful in most cases, it sometimes fails to perform crucial checks. Therefore, we choose to check Formula (5) also by Vampire [26]. Thanks to the wide adoption of the TPTP format, only minimal code changes are needed to try another prover.

We do not take any measures to request deterministic behaviour from the tools we use. This means that different runs can find different traps and, consequently, different PINs. Regardless, all the successes in the above table are robust (as illustrated by running ten successive computations of the same examples to obtain the data for Figure 3).

6 Conclusion

In this contribution we proved the undecidability of safety properties for a very restricted programming model. This suggests that programming models which can model classical mutual exclusion algorithms honestly, necessarily are Turing complete.

Regardless, we developed a semi-decision procedure which is aimed at honest models of mutual exclusion algorithm. Implemented as the tool heron, this procedure produces readable and verifiable proofs for mutual exclusion algorithms by generalizing inductive invariants of finite instances to inductive invariants of the parameterized system.

Moreover, these invariants are concise. Consequently, formulating them in first-order logic or, more specifically, in the widely adopted TPTP format, allows us to provide readable invariants for the parameterized system. Additionally, we can leverage the mature tools of automated theorem proving to discharge necessary proof obligations and provide externally verifiable explanations of our proofs.

In contrast to comparable approaches which use specifically tailored abstractions we are outgunned regarding computation time. However, we provide a more readable explanation of proofs. This gives transparency and high reliability of the established results since they can be externally verified.

7 Future Work

Since our work was initially inspired by the success of the proving procedure in [21] for mutual exclusion algorithms, and by the promising results regarding generalized notions of "traps" in parameterized systems [13] we focused our work on mutual exclusion algorithms. An expansion to general safety properties is a natural next step. Additionally, the logical embedding allows for more elaborate semantics; e.g., non-deterministic assignments and unbounded domains. For the latter though fixing N does not yield a finite state space for an instance. Thus, the method to prove finite instances has to be altered. Moreover, all negative examples in our benchmark always fail while proving finite instances of the parameterized system. Hence, different instantiations of the used framework of

proving finite instances, generalizing obtained explanations and proving them via first-order theorem proving can be explored.

Also we believe that exploring the limitations of logical invariants is an interesting area for further research. For example it is observed in [4] and [28] that proving the correctness of a non-atomic version of Szymanski's algorithm by abstraction requires to account for the existence of a "blocking" agent in certain configurations. We strongly believe that this renders correctness proofs of honest representations of Szymanski impossible for universally quantified inductive invariants. This raises the question if there are conceptual properties of algorithms which relate them to the syntactic logical class required (or sufficient) to prove them correct.

Data Availability Statement and Acknowledgements. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 787367 (PaVeS).
The tool `heron` with examples is available at https://doi.org/10.5281/zenodo.4088630. The most recent version is maintained under https://gitlab.lrz.de/i7/heron.

References

1. Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda, and Ahmed Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 721–736. Springer, 2007.
2. Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 145–157. Springer, 2007.
3. Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Approximated parameterized verification of infinite-state processes with global conditions. *Formal Methods Syst. Des.*, 34(2):126–156, 2009.
4. Parosh Aziz Abdulla, Frédéric Haziza, and Lukás Holík. Block me if you can! - context-sensitive parameterized verification. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, pages 1–17, 2014.
5. Parosh Aziz Abdulla, Frédéric Haziza, and Lukás Holík. Parameterized verification through view abstraction. *Int. J. Softw. Tools Technol. Transf.*, 18(5):495–516, 2016.
6. Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno, and Ahmed Rezine. Handling parameterized systems with non-atomic global conditions. In *VMCAI*, volume 4905 of *Lecture Notes in Computer Science*, pages 22–36. Springer, 2008.
7. Parosh Aziz Abdulla, A. Prasad Sistla, and Muralidhar Talupur. Model checking parameterized systems. In *Handbook of Model Checking*, pages 685–725. Springer, 2018.
8. Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jiazhao Xu, and Lenore D. Zuck. Parameterized verification with automatically computed inductive assertions. In *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, pages 221–234, 2001.

9. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.
10. Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-finder: A tool for compositional deadlock detection and verification. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 614–619. Springer, 2009.
11. Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. Decidability in parameterized verification. *SIGACT News*, 47(2):53–64, 2016.
12. Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer, 1997.
13. Marius Bozga, Javier Esparza, Radu Iosif, Joseph Sifakis, and Christoph Welzel. Structural invariants for the verification of systems with parameterized architectures. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I*, pages 228–246, 2020.
14. Marius Bozga, Radu Iosif, and Joseph Sifakis. Checking deadlock-freedom of parametric component-based systems. In *TACAS (2)*, volume 11428 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2019.
15. Yu-Fang Chen, Chih-Duo Hong, Anthony W. Lin, and Philipp Rümmer. Learning to prove safety over parameterised concurrent systems. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 76–83, 2017.
16. N. G. de Bruijn. Additional comments on a problem in concurrent programming control. *Commun. ACM*, 10(3):137–138, 1967.
17. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.
18. Edsger W. Dijkstra. *Cooperating Sequential Processes*, pages 65–138. Springer New York, New York, NY, 2002.
19. Murray A. Eisenberg and Michael R. McGuire. Further comments on dijkstra's concurrent programming control problem. *Commun. ACM*, 15(11):999, 1972.
20. E. Allen Emerson and Vineet Kahlon. Model checking guarded protocols. In *LICS*, pages 361–370. IEEE Computer Society, 2003.
21. Javier Esparza, Ruslán Ledesma-Garza, Rupak Majumdar, Philipp J. Meyer, and Filip Niksic. An smt-based approach to coverability analysis. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 603–619, 2014.
22. Javier Esparza and Stephan Melzer. Verification of safety properties using integer programming: Beyond the state equation. *Formal Methods Syst. Des.*, 16(2):159–189, 2000.
23. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014.
24. Warren D. Goldfarb. The unsolvability of the godel class with identity. *J. Symb. Log.*, 49:1237–1252, 1984.

25. Donald E. Knuth. Additional comments on a problem in concurrent programming control. *Commun. ACM*, 9(5):321–322, 1966.
26. Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 1–35, 2013.
27. Leslie Lamport. The mutual exclusion problem: part II - statement and solutions. In *Concurrency: the Works of Leslie Lamport*, pages 247–276. 2019.
28. Zohar Manna and Amir Pnueli. An exercise in the verification of multi-process programs. In W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty Is Our Business: A Birthday Salute to Edsger W. Dijkstra*, pages 289–301, New York, NY, 1990. Springer New York.
29. Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, pages 82–97, 2001.
30. Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.
31. Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In Pacal Fontaine, editor, *Proc. of the 27th CADE, Natal, Brasil*, number 11716 in LNAI, pages 495–507. Springer, 2019.
32. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.
33. Boleslaw K. Szymanski. Mutual exclusion revisited. In *Next Decade in Information Technology: Proceedings of the 5th Jerusalem Conference on Information Technology 1990, Jerusalem, October 22-25, 1990*, pages 110–117, 1990.
34. Gadi Taubenfeld. The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms. In *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, pages 56–70, 2004.
35. Klaus von Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. Cardinalities and universal quantifiers for verifying parameterized systems. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 599–613. ACM, 2016.
36. Christoph Welzel, Javier Esparza, and Mikhail Raskin. heron. https://doi.org/10.5281/zenodo.4088630, https://gitlab.lrz.de/i7/heron, October 2020.
37. Lenore D. Zuck and Kenneth L. McMillan. Invisible invariants are neither. In *From Reactive Systems to Cyber-Physical Systems - Essays Dedicated to Scott A. Smolka on the Occasion of His 65th Birthday*, pages 57–72, 2019.