Abstract

Parallel and distributed systems rely on intricate protocols to manage shared resources and synchronize, i.e., to manage how many processes are in a particular state. Effective verification of such systems requires universally quantification to reason about parameterized state and cardinalities tracking sets of processes, messages, failures to adequately capture protocol logic. In this paper we present #Π, an automatic invariant synthesis method that integrates cardinality-based reasoning and universal quantification. The resulting increase of expressiveness allows #Π to verify, for the first time, a representative collection of intricate parameterized protocols.

1. Introduction

Parallel and distributed systems rely on intricate protocols to manage shared resources and synchronize, i.e., to manage how many processes are in a particular state [12, 32, 40, 43]. Verification tools can support development of provably correct parallel and distributed systems by inferring and/or checking correctness arguments [23, 24, 39, 42, 50], while offering various degrees of expressiveness and automation. In this paper we aim at advancing expressiveness of fully automatic invariant synthesis for the verification of parallel and distributed systems.

Verifying systems parameterized by the number of executed processes requires dealing with global (system) states modeled by arrays of local (process) states. Universally quantified assertions over such arrays provide a commonly adopted and used language to symbolically represent parameterized state spaces, where quantifiers range over process identifiers. Tools for automatic inference of quantified array invariants can significantly reduce the manual annotation burden [1–4, 41, 49]. Unfortunately, universal quantifiers alone are not sufficient to express proofs of even modestly complex protocols.

Consider a lemma from a mutual exclusion proof for n processes extracted from a standard textbook [28, page 30]:

For $j$ between 0 and $n - 1$, there are at most $n - j$ threads at level $j$.

It imposes a symbolic bound on the cardinality of a particular set of threads, which is an essential protocol invariant. Similarly, many protocol descriptions and corresponding correctness proofs routinely refer to cardinality of sets of processes (messages, links, failures) [23, 35, 39, 56]. When verifying parallel and distributed systems, cardinality-based reasoning is just as important as universal quantification. Tools for automatic inference of what to count [17, 18] and tools that discover invariants over manually specified auxiliary counters [7, 19, 47] can contribute towards automation of cardinality-based verification. Unfortunately, these tools do not support universally quantified invariants, and can only deal with a finite collection of auxiliary counters. Note that even our simple textbook example refers to $n$ cardinalities, one for each level.

In this paper we present #Π, an invariant synthesis method that integrates cardinalities and universal quantification. #Π can synthesize invariants tracking relations between 1) scalars, 2) cardinalities of sets represented using predicates over scalars and arrays, and 3) universally quantified array assertions. This powerful combination facilitates fully automatic proofs of parameterized systems that were out of reach for automatic tools until now, as presented by examples in Section 7.

Our approach builds upon an observation that update statements in parameterized systems make only point-wise updates to the system state, i.e., just one thread moves at a time. We present an axiomatization of cardinality that is tailored to such updates. It allows #Π to reason about relations between cardinalities of sets defined by assertions over arrays by reducing reasoning about cardinalities to reasoning about quantified array assertions. This powerful combination facilitates fully automatic proofs of parameterized systems that were out of reach for automatic tools until now, as presented by examples in Section 7.

Our approach builds upon an observation that update statements in parameterized systems make only point-wise updates to the system state, i.e., just one thread moves at a time. We present an axiomatization of cardinality that is tailored to such updates. It allows #Π to reason about relations between cardinalities of sets defined by assertions over arrays by reducing reasoning about cardinalities to reasoning about quantified array assertions. In order to provide formal guarantees on the precision of our axiomatization, we show that our axiomatization of point-wise updates is relatively complete with respect to difference bound constraints.

We implemented #Π by relying on emerging Horn constraint solving technology [20, 29, 30, 33, 36, 51, 52]. We applied it on a collection of parameterized systems, including mutual exclusion, consensus, and garbage collection. The evaluation shows that #Π pursues a viable approach. It
efficiently synthesized expressive cardinality-based universally quantified invariants for intricate protocols. All but one of them were verified fully automatically for the first time. We observed that #Π can even outperform existing semi-automatic tools for parameterized verification that require manually specified counters.

In order to demonstrated that the ability of #Π to deal with cardinality does not incur any overhead when cardinality reasoning is not required, we compare #Π state-of-the-art tools on parameterized systems whose proofs are cardinality free. Our experiments show that #Π performs as least as well, and often better.

In summary, this paper contributes an automatic method for synthesizing cardinality-based universally quantified invariants of parallel and distributed systems together with its implementation and experimental evaluation.

2. Motivating examples

In this section, we discuss three examples that highlight different challenges in verifying parametrized protocols: combination of cardinalities and universal quantification, reasoning with array of counters, and reasoning with synchronous composition of processes.

Ticket lock  Figure 1 contains code for the classic ticket lock mutual exclusion protocol. This protocol makes use of a global ticket counter $t$ and a global service counter $s$. Whenever a thread wants to enter the critical section it draws a ticket by assigning $t$ to a local variable $m$. It then increments $t$ and spins until the service counter has reached the value of its previously drawn ticket stored in $m$. Upon leaving the critical section, the thread increments $s$ in order to allow the next thread to enter. For this example, we want to prove mutual exclusion, i.e. we want to show that the number of threads at location 3 is bounded by 1. For this, #Π synthesizes the following invariant which states that the number of threads that are either ready to enter the critical section or already inside the critical section is bounded by 1.

$$\#\{ t \mid m(t) \leq s \land pc(t) = 2 \} + \#\{ t \mid pc(t) = 3 \} \leq 1$$

Additionally, it discovers the following invariant stating that tickets are unique.

$$\forall t, t' : m(t) = m(t') \rightarrow t = t'$$

Despite its apparent simplicity, this example requires both quantification and cardinalities which highlights the fact that an automated method for verifying parametrized protocols needs to be able deal with both.

Filter lock  This example expands on the protocol discussed in the introduction. Figure 2 shows a code fragment that implements the filter lock, a well-known mutual exclusion protocol [28]. We model this protocol using a cardinality constraint, in line 5. The protocol is based on the following idea:

- There are $n-1$ “waiting rooms” called levels.
- Threads try to increase their level in order to acquire the lock, which corresponds to reaching level $n-1$.
- For each level, at least one thread trying to enter the level succeeds. This is guaranteed by the condition $\#\{ t \mid lv(t) > i \} = 0$ in the if-statement in line 5 that allows a thread to enter the next level if there are no threads at higher levels.
- If there are threads on higher levels, exactly one thread that enters a given level gets blocked, i.e. continues waiting at that level. This is enforced though the condition $\#\{ t \mid lv(t) = i \} \geq 2$ in line 5, which allows
The invariant states that the number of threads that have
most often received value. Finally, if more than two-thirds
more than two-thirds of the total number of processes
\(x\) value
\(r\) gets assigned a set of processes from which it received mes-
is a synchronous, round-based model where each processes
failures, (i.e., transmission-, but not Byzantine failures). This
algorithm in the heard-of model [13] which captures benign
one of the initial values as a common output. We specify the
and the goal of the protocol is for the processes to agree on
One-third rule
Figure 3 shows code for the
in our method.
unbounded number of cardinalities. This highlights the fact
that cardinalities and quantifiers cannot be treated in isola-
der a quantifier. This means that rather than keeping track of
appear in isolation, but the cardinality constraint shows up un-
der a quantifier. This means that rather than keeping track of
a fixed number of cardinalities, the method needs to track an
unbounded number of cardinalities. This highlights the fact
that cardinalities and quantifiers cannot be treated in isolation
but require a close integration such as the one provided
in our method.

**One-third rule** Figure 3 shows code for the *one-third*
rule [13, 14], which implements a simple consensus pro-
tocol. The protocol is executed by a number of processes,
where each process starts the protocol with an initial value \(v_0\),
and the goal of the protocol is for the processes to agree on
one of the initial values as a common output. We specify the
algorithm in the heard-of model [13] which captures benign failures, (i.e., transmission-, but not Byzantine failures). This
is a synchronous, round-based model where each processes
gets assigned a set of processes from which it received mes-
gles in a given round. For round \(r\), we denote this set by
\(HO(r)\).
A process starts a round by sending its local candidate
value \(x\) to all other processes. If it received messages from
more than two-thirds of the total number of processes \(n\), the
process updates its local candidate value \(x\) with the smallest,
most often received value. Finally, if more than two-thirds
of all processes sent the previously selected value \(x\) as their
candidate, the process decides on \(x\) by assigning it to \(res\).

# II automatically verifies the following properties of this
protocol:

• Agreement: whenever two processes have reached a de-
cision, the values they have decided on must be equal.

• (Weak) validity: if all processes propose the same initial
value, they must decide on that value.

• Irrevocability: if a process has decided on a value it does
not revoke its decision later.

To prove the above properties, our method synthesizes the
following invariant.

\[
\forall p : res(p) \geq 0 \rightarrow \#\{t \mid x(t) = x(p)\} > \frac{2n}{3} \\
\land x(p) = res(p)
\]

This invariant states that if a process has decided on a value
\(res\), then that value must be equal to its local candidate and
more than two-thirds of the processes must have proposed
the same value.

This example highlights the need to adress different
models of communication such as synchronous and asyn-
chronous communication. In our method, we achieve this by
relying on logic as a means to encode models rather then a
priori committing to a particular one.

3. Informal overview

In this section we illustrate the main ideas behind our
method through a simple example. Consider the following
program in which an unbounded number of threads incre-
ment a global variable \(a\) which is initialized to 0.

```
global int a;
1: a++;
```

The property we want to prove about this program is that
whenever there is a thread at location 2, variable \(a\) must be
larger that zero.

For this, we represent the program by the following log-
cal assertions representing initial states, transition relation,
and a safety property. We model the program counter \(pc\)
as an array, where each position in the array corresponds to the
program counter of a single thread. Assertion \(next\) uses \(t'\) to
denote the identifier of an arbitrary thread that increments \(a\).

```
init(a, pc) \equiv (\forall t : pc(t) = 1) \land a = 0

next(a, pc, a', pc') \equiv \exists t' : 
\begin{cases}
  pc(t') = 1 \land \\
  pc[pc[t' \leftarrow 2]] \\
  a' = a + 1
\end{cases}

safe(a, pc) \equiv (\exists t : pc(t) > 1) \rightarrow a > 0
```
The verification conditions are given by the following Horn constraints which ensure that $inv$ is a safe inductive invariant. We assume that each clause is implicitly universally quantified.

$$\exists inv(a, pc) :$$

(a) $init(a, pc) \rightarrow inv(a, pc)$

(b) $inv(a, pc) \land next(a, pc, a', pc') \rightarrow inv(a', pc')$

(c) $inv(a, pc) \rightarrow safe(a, pc)$

The following invariant is a solution to the above constraints. It states that $a$ is greater than the number of threads at position 2.

$$inv(a, pc) \overset{def}{=} \#\{t \mid pc(t) \geq 2\} \leq a$$

Finding such invariants automatically is our goal. However, for simplicity, we first show how such an invariant can be checked, if already given. We then show how our synthesis procedure discovers this invariant.

**Invariant checking** Checking validity of the above invariant (if already given) requires the ability to reason about cardinalities of sets defined over uninterpreted functions. In #II, we achieve this in a two-step process: in a first step, we replace applications of the cardinality operator by fresh variables, and in a second step instantiate cardinality axioms in order to regain lost information. We now describe this process for the above example.

For clause (a), we replace $\#\{t \mid pc(t) \geq 2\}$ by the fresh variable $k$, and instantiate an axiom stating that if $pc(t) \geq 2$ does not hold for any thread $t$, then the cardinality of the set defined by this predicate must by zero. Substituting and instantiating yields the following formula.

$$\left( \forall t : pc(t) = 1 \right) \land \left( \left( \forall t : pc(t) \leq 1 \right) \rightarrow k = 0 \right) \land a = 0 \rightarrow k \leq a$$

This formula contains universal quantification, however, since it falls into the array property fragment [11], the quantifiers can be eliminated. In order to prove validity for clause (b), we crucially need the ability to track how function updates affect cardinalities. We achieve this by instantiating an axiom an that relies on the following observation. An update $pc' = pc[t \leftarrow v]$ changes the function value of $pc$ only at position $t$. This means to track the overall effect of this update, it is enough to consider the changes at position $t$. In our example, updating the program counter from 1 to 2 moves a new thread into the set and hence the axiom strengthens the second clause with the formula $k' = k + 1$, where $k'$ is the fresh variable introduced for the cardinality after the update. For clause (c), we instantiate an axiom stating that, if there is at least one element in the set, the cardinality of the set is greater than zero.

Instantiation and quantifier elimination yields a quantifier and cardinality free formula whose validity can be efficiently checked by off-the-shelf SMT solvers.

**Invariant synthesis** To synthesise the above invariant, we restrict the search space to invariants of the following shape.

$$\left( \exists t : \left(s(pc(t), a) = k\right) \right) \land inv_0(pc, a, k)$$

This restriction requires the invariant to be composed of a set defined by an unknown predicate $s(pc(t), a)$ whose cardinality is bound to a variable $k$ and a cardinality-free part $inv_0(pc, a, k)$ which relates $k$ to other program variables. As in the checking case, our method removes all occurrence of the cardinality operator from the clauses and instantiates cardinality axioms. For clause (a) this yields

$$\left( \forall t : pc(t) = 1 \right) \land \left( \left( \forall t : \neg s(pc(t), a) \rightarrow k = 0 \right) \land \ldots \right) \rightarrow inv_0(pc, a, k)$$

where the dots represent additional omitted instances of cardinality axioms. Eliminating quantifiers yields

$$pc(t) = 1 \land \left( \neg s(pc(t), a) \rightarrow k = 0 \right) \land \ldots \rightarrow inv_0(pc, a, k) .$$

The resulting clauses are cardinality- and quantifier-free which allows us to apply existing Horn clause solvers. Passing the clauses to a solver returns the solution

$$s(pc(t), a) \overset{def}{=} \left(pc(t) \geq 2\right)$$

$$inv_0(pc, a, k) \overset{def}{=} \left(k \leq a\right).$$

**4. Preliminaries**

In this section, we define our notion of parametrized systems. We first discuss the asynchronous case. Let $l$ be a tuple of local variables, $g$ be a tuple of global variables, and $L$ denote a function that maps each thread identifier $t$ to a tuple of its local variables $l$. Then, a parametric system is given by three constraints: $init(g, L)$, $next_T(g, l, g', l')$, and $safe(g, L)$. Constraints $init(g, L)$ and $safe(g, L)$ define initial states and a safety property. These constraints can have arbitrary quantifier structure, however, cardinalities are restricted to occur in the quantifier-free part. Constraint $next_T(g, l, g', l')$ defines a local transition relation that describes how a single thread evolves the system. For this, it relates globals and locals to their primed versions, which represent the program state after the transition. We assume $next_T(g, l, g', l')$ to be quantifier-free.

Let $L[t \leftarrow l]$ denote the result of updating $L$ at position $t$ with $l$. Then, we define the global transition relation $next_T$ as follows.

$$next_T(g, L, g', L') \overset{def}{=} \exists t : \left( next_T(g, L(t), g', l') \land L' = L[t \leftarrow l'] \right)$$ (1)
This transition relation picks an arbitrary thread $t$, lets it evolve locals and globals, and finally updates the function $L$. The transition relation preserves locality in the sense that a thread can only update its own locals. We exploit this property in Section 5 where it enables tracking the influence of array updates on cardinalities.

For the synchronous case, where threads move in lock-step, the setting remains the same, however the quantifier in Equation 1 turns into a universal quantification.

We assume a standard semantics of computations, where a computation is defined as a sequence of states that starts from an initial state and is constructed by following the global transition relation. We say that a state is reachable, if and only if there exists a computation leading up to that state. A parametrized system is safe, if and only if only safe states are reachable.

The above definition allow us to apply a standard proof rule for safety to describe a safe, inductive invariant $\text{inv}(g, L)$ for the parameterized system. Since an instance of this proof rule is already shown in Section 3, here we only revisit that the invariant needs to 1) hold on initial states, 2) be inductive under the transition relation next and 3) imply the safety condition.

5. Cardinality Axioms

Consider the combined theory of linear arithmetic and arrays i.e., the theory of arithmetic extended with the interpreted functions $\cdot (\cdot)$ for array reads and $[\cdot ← \cdot]$ for array updates (see e.g. [11] for more details). Let $\varphi$ be a quantifier-free formula in that theory such that $\varphi$ does not contain the update function. Then, for variables $t$ and $k$ we call an expression $\#\{t \mid \varphi\} = k$ a cardinality constraint. In this paper, we consider the combined theory of linear arithmetic, arrays and cardinality constraints. We allow a restricted form of univ- ersal quantification over cardinality free formulas, such that a complete instantiation for the universal quantifiers can be efficiently computed (e.g. the array property fragment [11]). In order to avoid reasoning about infinite cardinalities, we assume that the set of all threads $\{t \mid \text{true}\}$ is of arbitrary but fixed size.

**Example 1.** The formulas

$$
(\forall t : f(t) = 1) \land \#\{t \mid f(t) \geq 2\} = k \land k \geq 1
$$

and

$$
\#\{t \mid f(t) = 2\} = k \land \#\{t \mid g(t) = 2\} = l \land f(j) = 1 \land g = f[j ← 2] \land l \leq k
$$

are formulas in the combined theory of arithmetic, arrays and cardinality constraints.

5.1 Elimination procedure

We now describe our instantiation procedure ELIMCARD which soundly eliminates cardinality constraints through a reduction to arithmetic and array reasoning. For a formula $\Delta$, our procedure first replaces all cardinality constraints by fresh variables, where the procedure maintains a bookkeeping function $\text{Def}(\cdot)$ that maps fresh variables to cardinalities. We assume that this function has a designated entry $\text{Def}(0) = \#\{t \mid \text{false}\}$ which represents the empty set. The procedure then instantiates a number of axioms that recover information about the previously eliminated cardinalities. Finally, ELIMCARD eliminates universal quantifiers, thus yielding a quantifier-, and cardinality-free formula whose validity can be checked by an SMT-solver.

Figure 4 shows rewriting rules for instantiating cardinality axioms.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Rule $\text{CARD}_\leq$.</td>
<td>$\text{Def}(k) = #{t \mid \varphi}$ $\text{Def}(l) = #{t \mid \varphi'}$ $\Delta$ $(\forall t : \varphi \to \varphi') \to k \leq l \land \Delta$</td>
</tr>
<tr>
<td>(b) Rule $\text{CARD}_&lt;$.</td>
<td>$\text{Def}(k) = #{t \mid \varphi(t)}$ $\text{Def}(l) = #{t \mid \varphi'(t)}$ $\Delta$ is conjunctive $g = f[j ← _ \text{ occurs in } \Delta]$ $\varphi' = \varphi[g/f]$ $\Delta$ $\cap (#(\varphi'(j), \delta^+) \cap #(\varphi(j), \delta^-) \cap (l = k + \delta^+ - \delta^-))$</td>
</tr>
</tbody>
</table>

Figure 4: Rewriting rules for instantiating cardinality axioms.
• Rule $\text{CARD}_< \leq$ instantiates a rule tracking inequalities between cardinalities.

• Rule $\text{CARD-UPD}$ models how cardinalities evolve through array updates. This rule makes use of the locality of parametric systems mentioned in Section 6.1, which ensures that each transition only updates one array entry at a time. This allows to characterize the effect of an array update on cardinality in the following way. When updating an array at position $j$, the cardinality of a set referring to $j$ is decremented if the $j$ was part of the set before the update, and incremented if $j$ is part of the set after the update. This is formalized through an indicator relation $\iota$. For a constraint $\varphi$ and variable $k$, we define $\iota$ as follows.

$$\iota(\varphi, k) \equiv (\varphi \land k = 1) \lor (\neg \varphi \land k = 0)$$

The rule $\text{CARD-UPD}$ can only be applied for formulas consisting of conjunctions, moreover the formulas $\varphi$ and $\varphi'$ that define the involved sets must be equivalent, except for the use of function $f$ and $g$ respectively. This condition ensures that the only difference in the cardinality of both sets stems from the function update.

**Example 1 (continued).** Consider again the formula

$$(\forall t : f(t) = 1) \land \#\{t \mid f(t) \geq 2\} = k \land k \geq 1 .$$

Let $\text{Def}(k) = \#\{t \mid f(t) \geq 2\}$, then, instantiating the axiom $\text{CARD}_< \leq$ for a comparison with the empty set yields the following formula

$$(\forall t : f(t) = 1) \land (f(t) \geq 2 \rightarrow \text{false}) \land k \geq 0 \land k \geq 1$$

which we simplify (for readability) into

$$(\forall t : f(t) = 1) \land (\exists t : f(t) \geq 2 \lor k \leq 0) \land k \geq 1 .$$

Instantiating the quantifiers produces the following equivalent formula that can be easily checked by an SMT solver.

$$f(t) = 1 \land (f(t) \geq 2 \lor k \leq 0) \land k \geq 1 .$$

For the formula

$$\#\{t \mid f(t) = 2\} = k \land \#\{t \mid g(t) = 2\} = l \land f(j) = 1 \land g = \text{func}(j \leftarrow 2) \land l \leq k$$

instantiating axiom $\text{CARD-UPD}$ yields

$$l = k + \delta^+ - \delta^- \land 1(f(j) = 2, \delta^+) \land 1(f(j) = 2, \delta^-) \land f(j) = 1 \land g = \text{func}(j \leftarrow 2) \land l \leq k$$

which simplifies to

$$l = k + 1 \land f(j) = 1 \land g = \text{func}(j \leftarrow 2) \land l \leq k$$

**Soundness** Our axioms are sound, which in turn underpins the soundness of $\Pi$.

**Theorem 1 (Soundness).** Axioms $\text{CARD}_< \leq$, $\text{CARD}_< \geq$, and $\text{CARD-UPD}$ are sound, i.e. the assertion under the line in Figure 4(a,b,c) is a logical consequence of $\Delta$.

**Derived Properties of $\text{CARD}_< \leq$ and $\text{CARD}_< \geq$** The following useful properties follow from Axioms $\text{CARD}_< \leq$ and $\text{CARD}_< \geq$.

- **CARD$_{\geq 0}$**: cardinalities are always non-negativ. That is for $k \in \text{dom}(\text{Def})$ we have $k \geq 0$.

- **CARD$_{> 0}$**: if there is no element in a set, the cardinality of that set is zero. For $\text{Def}(k) = \#\{t \mid \varphi\}$ the following holds.

$$\forall t : \neg \varphi \rightarrow k = 0$$

- **CARD$_{> 0}$**: if there is at least one element in a set, the cardinality of that set is greater than zero. That is for $\text{Def}(k) = \#\{t \mid \varphi\}$ the following holds.

$$\exists t : \varphi \rightarrow k > 0$$

**Relative completeness of $\text{CARD-UPD}$** We now prove that the update axiom preserves difference bound constraints. A difference bound constraint, is a conjunction of inequalities of the form $k \leq l + c$, where $c$ is a numeric constant and $k$ and $l$ are variables. The following theorem states that instantiating the axiom $\text{CARD-UPD}$ preserves difference bound constraints over cardinalities.

**Theorem 2 (Relative completeness $\text{CARD-UPD}$).** Let $\Delta$ be an arbitrary formula in the combined theory of cardinality constraints, arrays and arithmetic. We let $\Psi$ denote a formula containing the cardinality of two sets related through an update statement.

$$\Psi \equiv (\#\{t \mid \varphi\} = k) \land (\#\{t \mid \varphi'\} = l) \land g = \text{func}(j \leftarrow \_ \leftarrow) \land \Delta$$

where $\varphi' = \varphi[g/f]$. Let $\theta$ denote the same formula after the instantiation of the update axiom.

$$\theta \equiv (l = k + \delta^+ - \delta^-) \land 1(\varphi', \delta^+) \land 1(\varphi, \delta^-) \land \Delta$$

Then, if $\Psi$ is satisfiable, the following holds for all difference bound constraints $\rho(k,l)$.

$$\Psi \rightarrow \rho(k,l) \quad \text{if and only if} \quad \theta \rightarrow \rho(k,l)$$

For the proof of Theorem 2, we make use of the following proposition stating that equality constraints are maximal in the following sense: whenever an arbitrary formula implies an equality constraint, this equality constraint implies all difference bound constraints that are consequences of the formula.
Proposition 1. For all $\Psi$ such that $\Psi$ is satisfiable formula in any theory that includes arithmetic, and for all difference constraints $\rho(k,l)$ and constants $c$, if

$$\Psi \rightarrow l = k + c \text{ and } \Psi \rightarrow \rho(k,l)$$

hold then $l = k + c \rightarrow \rho(k,l)$.

Proof 1 (Theorem 2). The “right-to-left” direction follows from the fact that $\Psi \rightarrow \theta$ holds. For the “left-to-right” direction assume that $\Psi \rightarrow \rho(k,l)$ and $\theta$ hold, then we need to show $\rho(k,l)$. By case splitting over truth valuations for $\varphi$, and $\varphi'$, we get $\theta \rightarrow l = k + c$, for some $c$. Then, from $\Psi \rightarrow \theta$, we can deduce that $\Psi \rightarrow l = k + c$, and by Proposition 1, we get that $l = k + c \rightarrow \rho(k,l)$ from which $\rho(k,l)$ follows. □

Remark 1. If for all cardinalities $\#\{t \mid \varphi\}$, we restrict occurrences of $t$ in $\varphi$ to direct array reads, all axiom instantiations fall into the array-property fragment, and we can therefore efficiently compute a complete instantiation for universal quantifiers. We note that this is the case for all our examples.

5.2 Venn decomposition

While for all examples from the literature (i.e., those in Figure 7 and the upper table in Figure 6), the above axioms are sufficient, some examples (i.e., those in the lower table in Figure 6– in these examples comparison between cardinalities go beyond order constraints), require a form of Venn decomposition. For this, we assume that all cardinality constraints are of the form $\#\{t \mid \varphi\} = k$, where $\varphi$ is conjunctive (this applies to all inferred sets in our examples). Let $P$ denote the set of predicates (conjuncts) occurring in set comprehensions. Then, we decompose the universal set into regions corresponding to truth valuations of these predicates. For this purpose, we associate with each set $Q \in 2^P$ a region $\text{region}(Q)$, which we define as follows.

$$\text{region}(Q) \overset{\text{def}}{=} \{t \mid \bigwedge_{p \in Q} p \land \bigwedge_{p \in (P \setminus Q)} \neg p\}$$

Then, for each predicate $p \in P$, we add the following equation.

$$\#\{t \mid p\} = \sum\{ \#\text{region}(Q) \mid Q \in 2^P \text{ and } p \in Q \}$$

Finally, we add a decomposition of the universal set $\Omega \overset{\text{def}}{=} \{t \mid \text{true}\}$ through the following equation.

$$\#\Omega = \sum\{ \#\text{region}(Q) \mid Q \in 2^P \}$$

Example 2. Consider the following constraint, which illustrates an argument in the verification of the one-third protocol presented in Section 2. This constraint is unsatisfiable, however the axioms of Section 5.1 are not strong enough to derive a contradiction.

$$\#\{t \mid f(t) = 1\} \geq 2n \land \#\{t \mid g(t) = 1\} \geq 2n \land \#\Omega = n \land \#\{t \mid f(t) = 1 \land g(t) = 1\} = 0$$

The set of predicates is give by $P \overset{\text{def}}{=} \{f(t) = 1, g(t) = 1\} \overset{\text{def}}{=} \{a, b\}$. The Venn-decomposition produces the following equations.

$$\#\{t \mid a\} = \#\{t \mid a \land \neg b\} + \#\{t \mid a \land b\}$$
$$\#\{t \mid b\} = \#\{t \mid \neg a \land b\} + \#\{t \mid a \land b\}$$
$$\#\Omega = \#\{t \mid a \land \neg b\} + \#\{t \mid \neg a \land \neg b\}$$

From these equations, and the facts that $\#\{t \mid a \land b\} = 0$, and $\#\Omega = n$ we can derive the following equality.

$$n = \#\{t \mid a\} + \#\{t \mid b\} + \#\{t \mid \neg a \land \neg b\}$$

Then from $\#\{t \mid a\} \geq \frac{2n}{3} \land \#\{t \mid b\} \geq \frac{2n}{3}$ the contradiction follows. □

6. The method #I

In this section, we describe of our method #I which computes invariants for parametric systems by computing a solution to a set of Horn clauses in the combined theory of arithmetic, arrays and cardinalities. Our method relies on the following main steps.

- **Defining the search space** In this step, we restrict the search space for the invariant. For this, we provide a shape template which specifies the number of sets whose cardinality the invariant may refer to, as well as the number of quantifiers used in the invariant (Section 6.1).

- **Quantifier elimination** We then eliminate universal quantifiers that occur in the invariant. For this, we rely on existing methods [10, 31].

- **Cardinality elimination** In this step, we eliminate cardinalities from the clauses. For this, we replace all occurrences of cardinalities by fresh variables and recover relations between the freshly introduced variables by instantiating axioms as described in Section 5.

- **Solving** Finally, we rely on existing solvers (such as [9, 20, 29, 33]) to compute a solution for the resulting clauses. This yields the desired invariant.

6.1 Defining the search space

In order to define a search space for invariants, we require the user to provide a shape that fixes the number of cardinality expressions and universal quantifiers that are allowed to occur in the invariant. For an invariant $\text{inv}$ with $n$ quantifiers and $m$ cardinality expressions, this defines an assertion $\text{Shape}(\text{inv})$ of the following form, where $\text{inv}_0$ is an unknown quantifier-free assertion that relates cardinalities with program data, and $s_1, \ldots, s_m$ are unknown assertions defining the respective sets.

$$\forall q_1, \ldots, q_n : \#\{t \mid s_1\} = k_1 \land \cdots \land \#\{t \mid s_m\} = k_m \land \text{inv}_0$$
Algorithm #Π
input $C$, $Q$, Shape
output $Σ$ – solution function
local
    function ELIMCARD – Cardinality elimination
    functions INSTQ – Quantifier instantiation
    functions SOLVE – Horn clause solver
begin
1:    foreach $p ∈ dom(Shape)$ and $c ∈ C$ do
2:        $c ← c[Shape(p)/p]$
3:    $c ← INSTQ(Shape(p), c)$
4:    $C ← ELIMCARD(C)$
5: return SOLVE($C$, $Q$)
end

Figure 5: Algorithm #Π.

Example 3. In the filter-lock example, we search for an invariant with 1 quantifier and 1 cardinality expression defining an expression $\text{Shape}(\text{inv}) ≜ ∀q : \#\{t | s\} \land inv_0$. ■

6.2 Algorithm #Π

Figure 5 shows method #Π. Its input is a set of clauses $C$, a set of existentially quantified predicates $Q$ that we refer to as queries and a shape template function $\text{Shape}$. #Π returns a solution function $Σ$ that maps each query to a constraint such all clauses in $C$ are valid if one substitutes each query by its solution. Function $\text{INSTQ}(\psi, c)$ takes as input a quantified formula $\psi$ and a clause $c$. It produces an instantiated clause as output. Function $\text{ELIMCARD}(C)$ takes as input a set of clauses and produces a set of cardinality-free clauses using the procedure described in Section 5.

The algorithm starts by plugging in shape templates for queries, and instantiating the universal quantifiers in the templates in lines 1–4 using function $\text{INSTQ}$. It then invokes function $\text{ELIMCARD}$. The resulting clauses are passed to a Horn solver, which returns a solution function.

7. Evaluation

In this section we evaluate our method which we have implemented in a research prototype #Π. We use a 1.3 Ghz Intel Core i5 computer with 4 GB of RAM for our experiments.

7.1 Cardinality-based reasoning

Table 6 summarises our results for reasoning with cardinalities. We use templates that specify the required number of quantifiers and set comprehension predicates.

Examples from [18] The upper table shows result on examples taken from [18]. We are not able to compare timings as, to the best of our knowledge, the technique has not yet been implemented. The examples consist of a simple running example intro, a simplified version of a Bluetooth device driver bluetooth, and a tree traversal routine tree traverse. The Bluetooth driver consists of a single stopping thread and an arbitrary number of worker threads. The property we prove is that whenever a worker thread is still active, the stopping process has not yet been completed. For the tree traversal example, we found that a simple invariant containing one universal quantifier is enough to prove the intended property.

Case studies The example cache consists of a simple model of a cache-coherence protocol taken from [55], for which we prove mutual exclusion. This is enforced by a cardinality constraint requiring that the critical section contains at most one thread. The lower part of Table 6 contains the case studies from Section 2. We note that the ticket example $^2$ from [1] is a simplification of our example as their formulation contains universally quantified guards in the transitions system which allows a direct encoding of the fact that a ticket is minimal among all threads. Farzan et al. analyze the same example in [17], however, it is not possible to express mutual exclusion directly in their formalism which requires proving a stronger property from which mutual exclusion follows via a manual argument.

Garbage collection The benchmark garbage collection, consists of a simple model of a tri-colour mark-and-sweep garbage collector for which we provide code in Figure 8. This garbage collector partitions memory locations (nodes) into three disjoint sets: black nodes that are reachable and hence in use, white nodes that are candidates for deletion, and grey nodes that are known to be reachable but whose descendants have not yet been marked. The algorithm proceeds by picking a node in the grey set, marking all its successors as grey, and finally moving the node into the black set. If the grey set is empty, all white nodes are unreachable and can be deleted. We model this algorithms through an arbitrary number of mutator-threads (function ArrWrite) that non-deterministically move nodes from the white into the grey set, and a single marker-thread (function ArrMark) that first sweeps the address space to non-deterministically move nodes from the white into the grey set (which models exploring successors), and in a second pass moves all nodes from the grey into the black set. Access to the nodes is regulated through a simple lock.

An important invariant of this algorithm is that nodes can only be set to a darker colour, i.e., once a node has been shown to be reachable, it cannot be re-considered for elimination. We model this property via an auxiliary variable. Proving monotonicity depends on the fact that mutual exclusion between mutators and sweeper thread is maintained.

$^2$ For ticket, we represented the uniqueness constraint of Section 2 through a cardinality constraint stating that for each ticket, there is at most one thread holding it.
### Table 6: Applying #II to cardinality-based reasoning.

<table>
<thead>
<tr>
<th>Program</th>
<th>Card</th>
<th>Correct</th>
<th>Property</th>
<th>Inferred cardinalities</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>intro [17]</td>
<td>✓</td>
<td>✓</td>
<td>∀t : pc(t) = 5 → prev ≤ max</td>
<td># {t</td>
<td>\text{pc}(t) ≤ 2}, # {t</td>
</tr>
<tr>
<td>bluetooth [17]</td>
<td>✓</td>
<td>-</td>
<td>∀t : pc(t) = 5 → prev ≤ max</td>
<td>-</td>
<td>7.2s</td>
</tr>
<tr>
<td>tree traverse [17]</td>
<td>✗</td>
<td>✗</td>
<td>∀t : \text{pc}(t) = 3 → alloc = 1</td>
<td># {t</td>
<td>\text{pc}(t) ≤ 3}, # {t</td>
</tr>
<tr>
<td>cache [55]</td>
<td>✓</td>
<td>✓</td>
<td>∀t : leaves = nodes + 1</td>
<td>-</td>
<td>4.2s</td>
</tr>
<tr>
<td>garbage collection</td>
<td>✓</td>
<td>✓</td>
<td>∀t : {2 ≤ \text{pc}(t) ≤ 4}</td>
<td># {t</td>
<td>2 ≤ \text{pc}(t) ≤ 4}</td>
</tr>
</tbody>
</table>

### Figure 6: Applying #II to cardinality-based reasoning. The column Card indicates whether or not cardinalities were used in the proof. Except ticket lock, all examples were automatically verified for the first time.

### Table 7: Comparison with benchmarks taken from [19]. The benchmarks consist of a number of simple barriers and locks. For each example, the benchmarks contain a buggy version of the example, where barriers have been removed, or other bugs have been introduced. We run buggy benchmarks with the same templates that were used to prove correctness of the original program. The comparison with timings taken from [19] shows that our method outperforms [19] on all examples. We attribute this to the fact that #II automatically discovers what to count rather than tracking cardinalities for all possible program locations.

<table>
<thead>
<tr>
<th>Program</th>
<th>Card</th>
<th>Correct</th>
<th>Property</th>
<th>Inferred cardinalities</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>max [19]</td>
<td>✓</td>
<td>✓</td>
<td>∀t : pc(t) = 5 → prev ≤ max</td>
<td># {t</td>
<td>\text{pc}(t) ≤ 2}, # {t</td>
</tr>
<tr>
<td>max-nobar [19]</td>
<td>-</td>
<td>✗</td>
<td>∀t : pc(t) = 5 → prev ≤ max</td>
<td>-</td>
<td>7.2s</td>
</tr>
<tr>
<td>reader/writer [19]</td>
<td>✗</td>
<td>✓</td>
<td>readcount &gt; 0 → writing = -1</td>
<td># {t</td>
<td>\text{pc}(t) ≤ 3}</td>
</tr>
<tr>
<td>reader/writer-bug [19]</td>
<td>-</td>
<td>✗</td>
<td>readcount &gt; 0 → writing = -1</td>
<td>-</td>
<td>0.8s</td>
</tr>
<tr>
<td>parent/child [19]</td>
<td>✓</td>
<td>✓</td>
<td>∀t : pc(t) = 3 → alloc = 1</td>
<td># {t</td>
<td>2 ≤ \text{pc}(t) ≤ 3}</td>
</tr>
<tr>
<td>parent/child-nobar [19]</td>
<td>-</td>
<td>✗</td>
<td>∀t : pc(t) = 3 → alloc = 1</td>
<td>-</td>
<td>3s</td>
</tr>
<tr>
<td>simp-bar [19]</td>
<td>✓</td>
<td>✓</td>
<td>∀t : pc(t) = 5 → fl = 1</td>
<td># {t</td>
<td>\text{pc}(t) ≤ 3}, # {t</td>
</tr>
<tr>
<td>simp-nobar [19]</td>
<td>-</td>
<td>✗</td>
<td>∀t : pc(t) = 5 → fl = 1</td>
<td>-</td>
<td>1.8s</td>
</tr>
<tr>
<td>dynamic-barrier [19]</td>
<td>✗</td>
<td>✓</td>
<td>∀t : \text{pc}(t) ≤ 2 → fl = 1</td>
<td># {t</td>
<td>\text{pc}(t) ≤ 3}, # {t</td>
</tr>
<tr>
<td>dynamic-barrier-nobar [19]</td>
<td>-</td>
<td>✗</td>
<td>∀t : \text{pc}(t) ≤ 2 → fl = 1</td>
<td>-</td>
<td>1.3s</td>
</tr>
<tr>
<td>as-many [19]</td>
<td>✓</td>
<td>✓</td>
<td>c_1 = c_2</td>
<td># {t</td>
<td>\text{pc}(t) ≥ 2}</td>
</tr>
<tr>
<td>as-many-bug [19]</td>
<td>-</td>
<td>✗</td>
<td>c_1 = c_2</td>
<td>-</td>
<td>0.7s</td>
</tr>
</tbody>
</table>

### Figure 7: Comparison to [19], [19] maintains counters for each possible program location rather then inferring what to count. The column Correct indicates whether or not the program meets is specification. #II outperforms [19] on all examples. We attribute this to the fact that #II infers a suitable subset of relevant cardinalities.

Hence, this example highlights that our method can efficiently deal with the interplay of safety properties and cardinalities.

### Comparison with [19]

Table 7 contains the results of a comparison with benchmarks taken from [19]. The benchmarks consist of a number of simple barriers and locks. For each example, the benchmarks contain a buggy version of the example, where barriers have been removed, or other bugs have been introduced. We run buggy benchmarks with the same templates that were used to prove correctness of the original program. The comparison with timings taken from [19] shows that our method outperforms [19] on all examples. We attribute this to the fact that #II automatically discovers what to count rather than tracking cardinalities for all possible program locations.

### 7.2 Cardinality-free reasoning

The ability to synthesize quantified invariants allows us to handle examples of cardinality-free reasoning from the literature. We compare #II to the methods from [1] and [49]. Table 9 summarises the results. Benchmarks in [1] consist of a number of mutual-exclusion protocols that require in-
global Lock L;
void ArrWrite(int addr) {
1: acquire(L);
2: if (ArrC(addr) == WHITE)
3: ArrC(addr) = GRAY;
4: release(L);
}
void ArrMark()
1: addr = lo;
2: while (addr < hi) {
3: acquire(L);
4: if (ArrC(addr) == WHITE)
5: ArrC(addr) := GRAY;
6: release(L);
7: addr = addr+1;
8: }
9: addr := lo;
10: while (addr < hi) {
11: acquire(L);
12: if (ArrC(addr) == GRAY)
13: ArrC(addr) = BLACK;
14: release(L);
15: addr = addr+1;
}
}

Figure 8: Code for the benchmark garbage collection.

variants with two universal quantifiers. In our experiments, we provide templates that specify the number of required quantifiers (only). We find that #Π performance is on par with [1] when using a solver over the reals and slightly faster when solving over integers. Examples from [49] consist of two variants of memory barrier implementations, a work stealing algorithm for processing arrays, the dining philosophers protocol, and a model of robot swarm on a fixed-sized grid. Columns I, P, and O, show timings from [49] for interval, polytope and octagon domains, respectively. Sanchez et al. provide timings for several abstraction schemes, however, we show only timings from the interference abstraction scheme as these are most favorable. We observe that #Π is out-performed by the interval abstraction, however, its performance is on par with the polytope domain, and scales better that the octagon domain. The reduced performance with respect to the interval domain can be seen as the penalty of generality since our method can find invariants consisting of arbitrary, (disjunctive) linear arithmetic formulas.

Figure 9: Cardinality-free reasoning: Results of comparing #Π to [1] and [49]. The column Q shows the number of universal quantifiers in the synthesized invariant.

8. Related Work

We broadly divide the related work into logics that support cardinality reasoning, verification methods for parameterized systems that rely on cardinality arguments, and methods that rely on universally quantified invariants. The main contribution of #Π in comparison with the following methods is the ability to reason about and synthesize assertions that combine cardinality with universal array assertions.

Quantitative Logics

The logic of Boolean algebra and Presburger arithmetic (BAPA) is studied in [37] and generalized to multi-sets and fractional collections in [44, 45] and direct and inverse function/relation images in [54]. This logic is however not suitable for our purposes, as sets are uninterpreted. Hence the logic cannot be used for reasoning about set which are explicitly defined through predicates over the program state, such as \( \{ t \mid pc(t) \geq 2 \} \). The examples we considered require this ability when constructing invariants.

[53] proposes a method for quantitative interpolation in the theory of linear arithmetic and employs this method for the verification of programs encoded as Horn constraints. In contrast, our method focuses on the treatment of uninterpreted functions which are used to encode the state of individual processes in the parametric system.

Dragó et al. propose a logic that contains cardinality constraints over uninterpreted functions as well as limited quantifier alternation in [14]. This logic is geared towards the verification of consensus protocols such as Paxos [38] in the heard-of model [13] which allows for benign (communication) faults. While the logic is similar in spirit to our approach, [14] focuses on satisfiability checking in an ex-
pressive subset of first order logic with the primary intent of checking inductive correctness arguments, whereas our focus lies on synthesizing such arguments automatically.

[19] presents a logic that allows assertions on the number of threads that are at a particular program location. The paper presents a verification method for this logic that relies on predicate-, and counter-abstraction, however, the method does not take into account universal quantification.

The abstract interpretation based approach presented in [22] can track memory partition sizes to infer memory usage properties. It relies on size tracking domain operations and can reason about data structures domains. An extension of such operations with tracking quantified array properties could lead to a viable alternative to the direct axiomatization.

**Quantitative verification of parametric systems** A classic example of the use of quantitative abstractions for parametric system is [46], where a number of bounded auxiliary counters for predefined sets of states are used to prove liveness of parametric protocols. The CIRC extension [26] of Blast [25, 27] shows how auxiliary counters could be inferred under predicate abstraction. [8] shows how counter updates can be inserted in a context-dependant way during model checking thus reducing the burden of tracking large numbers of cardinalities. Our method avoids the need to track large numbers of a priori defined cardinalities by automatically synthesizing descriptions of the required sets. Moreover, these methods do not support the combination of cardinalities with quantifiers.

Recently, Farzan et. al [17] proposed a method to infer auxiliary counters which they formalized in the framework of counting automata, and which they employed in the context of verifying parametric systems. This method is based on an encoding of conditions on a suitable counting automaton as an SMT problem over arithmetic and uninterpreted functions. In contrast, our method directly refers to cardinalities of (defined) sets, and thus avoids reasoning about auxiliary variables. Moreover [17] does not support the combination of counters with universal quantification.

**Qualitative verification of parametric systems** We now discuss methods for cardinality-free reasoning about parametric systems and limit ourselves to methods over infinite domains. The invisible invariants method relies on small instantiation to generate candidates for universally quantified array invariants and proposes fragments where checking these candidates can be done effectively [5, 15] even in the presence of complex communication topologies [6]. Our approach computes quantifier instantiation as a part of the inference process. In [34] the authors introduce inter-thread predicates that can express dependencies between the local variables of one thread and all local variables of another thread together with a mechanism to ensure monotonicity of boolean programs that arise from computing an abstraction with such predicates. This allows them to express properties such as: “variable m of all other threads” which enables verifying the ticket lock. In contrast, our method can often avoid tracking such dependencies by referring to the cardinality of the set of threads at a given location. [49] proposes the notion of reflective abstractions. In this framework, a proof is constructed by instantiating the transition system with a finite number of threads and modeling the effect of the remaining threads through a mirror thread. The method then uses abstract interpretation to infer an invariant for the instantiated system. [1] introduces a formalism that allows to express global conditions which relate local variables of different threads, and uses backward reachability to verify safety properties. The use of data flow graphs in [16] allows to separate reasoning about data and control and thus enables inferring invariants that holds for arbitrary many threads. Our approach relies on transition relations, however it may be interesting to adopt the data flow graph perspective in our setting.

9. Conclusion

Parameterized systems model core protocols of software infrastructures. Their verification often resorts to cardinality-based arguments as a concise and effective reasoning tool. Unfortunately, the problem of automatic inference of cardinality-based invariants was under-studied and viable tool support is scarce. This paper presented #II, a method and an implementation for the automatic inference of invariants that track cardinalities of assertions in the combined theory of scalars and arrays under universally quantified constraints. The axiomatization of cardinality we devised for #II yielded an effective tool that is capable of verifying intricate parameterized systems using cardinality arguments, going beyond possible with the state-of-the-art. At the same time #II is competitive or even outperforms the existing verifiers for parameterized systems that do not require cardinality arguments.

As of today, our approach has the following main limitations, which we consider callenges for future work.

- We do not consider heap allocated data structures. (Universal quantification in #II could provide some information, following [21], but this is currently not explored.)
- We do not investigate the effectiveness of #II for modular reasoning in the presence of procedures. (Targeting the case when procedures coincide with transactions [48] appears to be a promising direction to consider.)
References


