Cardinalities and Universal Quantifiers for Verifying Parameterized Systems

Klaus v. Gleissenthall
Technische Universität München, Germany and University of California, San Diego, USA
gleissen@ucsd.edu

Nikolaj Bjørner
Microsoft Research, USA
nbjorner@microsoft.com

Andrey Rybalchenko
Microsoft Research, USA
rybal@microsoft.com

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 universal 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.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; C.2.4 [*Computer-Communication Networks*]: Distributed Systems

Keywords Concurrency, verification, parametric systems, cardinalities

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 [Burrows 2006; Hunt et al. 2010; Ongaro and Ousterhout 2014; Mic 2015]. Verification tools can support development of provably correct parallel and distributed systems by inferring and/or checking correctness arguments [Newcombe et al. 2015; Hawblitzel et al. 2015b; Lamport 2015; Hawblitzel et al. 2015a; Sergey et al. 2015], 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 [Abdulla et al. 2007; Monniaux and Alberti 2015; Sanchez et al. 2012; Alberti et al. 2014a,b, 2015]. 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 [Herlihy and Shavit 2008, 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 or failures) [Kotla et al. 2007; Lamport 2015; Hawblitzel et al. 2015a; Zave 2015]. When verifying parallel and distributed systems, cardinality-based reasoning is just as important as universal quantification. Tools for automatic inference of what to count [Farzan et al. 2014] and tools that discover invariants over manually specified auxiliary counters [Basler et al. 2009a; Ganjei et al. 2015; Pnueli et al. 2002b] can contribute towards automation of cardinality-based verification. Unfortunately, these tools do not support universally quantified invariants, and can only

PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA
ACM. 978-1-4503-4261-2/16/06...$15.00
http://dx.doi.org/10.1145/2908080.2908129

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,[1] 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. 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 [Grebenshchikov et al. 2012; Hoder and Bjørner 2012; Kahsai et al. 2015; Hojjat et al. 2012; Kroening and Lewis 2014; Terauchi and Unno 2015; Unno and Terauchi 2015]. 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 demonstrate that the ability of #Π to deal with cardinality does not incur any overhead when cardinality reasoning is not required, we compare #Π to 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.

[1] Pronounced as "sharpie".

```
ticket lock
  global int t = 0;
  global int s = 0;
  local m = −1;
  void lock() {
1:  atomic {m = t; t = t + 1;}
2:  while (m > s){}
3: }
4: void unlock() {
5:  if (m ≤ s)  {s = s + 1;}
6: }
end
```

Figure 1: Ticket lock.

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.

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

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 [Herlihy and Shavit 2008]. 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 $n - 1$.

```
    global int n;
    global int [] lv;
    assume (n ≥ 2);
    void lock() {
1:    local int me = ThreadID.get();
2:    local int i = 0;
3:    while (i < n − 1) {
4:      atomic {
5:        if (#{t | lv(t) > i} = 0 || #{t | lv(t) = i} ≥ 2) {
6:          i++; lv(me) = i;
7:        }
8:      }
9:    }
10:   }
```

Figure 2: Filter lock.

- 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 a thread to raise its level only if there is at least one other thread at its current level.
- Since n threads participate in the protocol, at most one thread at a time can reach level $n - 1$, which ensures mutual exclusion.

#Π automatically synthesizes the following quantified invariant which formalizes this argument.

$$\forall l : 0 \leq l \leq n - 1 \rightarrow \#\{t \mid lv(t) \geq l\} \leq n - l$$

This invariant states that the number of threads that have reached a given level l is bounded by $n - l$. This implies that there is at most one thread at level $n - 1$, from which the mutual exclusion property follows.

In this example, cardinalities and quantifiers do not appear in isolation, but the cardinality constraint shows up under 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 [Drăgoi et al. 2014; Charron-Bost and Schiper 2009],

```
    protocol oneThird
1:    instantiation x := v0 with v0 ≥ 0 ; res := −1;
2:    round r:
3:      send x to all processes
4:      if #HO(r) > 2n/3 then
5:        x = the smallest most often received value
6:        if more than 2n/3 values rec. equal x then
7:          res = x;
      end
    end
```

Figure 3: One-third rule consensus protocol.

which implements a simple consensus protocol. The protocol is executed by a number of processes, where each process starts the protocol with an initial value v_o 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 [Charron-Bost and Schiper 2009] 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 messages 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.

#Π automatically verifies the following properties of this protocol:

- Agreement: whenever two processes have reached a decision, 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.

$$\forall p : res(p) \geq 0 \rightarrow \#\{t \mid x(t) = x(p)\} > \tfrac{2n}{3} \\ \wedge\, x(p) = res(p)$$

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 address different models of communication such as synchronous and asynchronous 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 increment a global variable a which is initialized to 0.

```
      global int a;
1:    a++;
2:
```

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 logical 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.

$$\begin{aligned}
&\mathit{init}(a, pc) && \stackrel{\text{def}}{=} (\forall t : pc(t) = 1) \wedge a = 0\\
&\mathit{next}(a, pc, a', pc') && \stackrel{\text{def}}{=} \exists t' : \begin{pmatrix} pc(t') = 1 \wedge \\ pc' = pc[t' \leftarrow 2] \wedge \\ a' = a + 1 \end{pmatrix}\\
&\mathit{safe}(a, pc) && \stackrel{\text{def}}{=} (\exists t : pc(t) > 1) \rightarrow a > 0
\end{aligned}$$

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.

$$\begin{aligned}
&\exists \mathit{inv}(a, pc) :\\
&(a) \quad \mathit{init}(a, pc) && \rightarrow \mathit{inv}(a, pc)\\
&(b) \quad \mathit{inv}(a, pc) \wedge \mathit{next}(a, pc, a', pc') && \rightarrow \mathit{inv}(a', pc')\\
&(c) \quad \mathit{inv}(a, pc) && \rightarrow \mathit{safe}(a, pc)
\end{aligned}$$

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

$$\mathit{inv}(a, pc) \quad \stackrel{\text{def}}{=} \quad \#\{t \mid pc(t) \geq 2\} \leq a$$

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 #Π, 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.

$$\begin{aligned}
&(\forall t : pc(t) = 1)\\
&\wedge ((\forall t : pc(t) \leq 1) \rightarrow k = 0)\\
&\wedge a = 0\\
&\rightarrow k \leq a
\end{aligned}$$

This formula contains universal quantification, however, since it falls into the array property fragment [Bradley et al. 2006], 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 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 that 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.

$$(\#\{t \mid s(pc(t), a)\} = k) \wedge \mathit{inv}_0(pc, a, k)$$

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 $\mathit{inv}_0(pc, a, k)$ 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

$$\begin{aligned}
&(\forall t : pc(t) = 1) \wedge\\
&\wedge ((\forall t : \neg s(pc(t), a)) \rightarrow k = 0) \wedge \ldots\\
&\rightarrow \mathit{inv}_0(pc, a, k)
\end{aligned}$$

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

$$\begin{aligned}
&pc(t) = 1 \wedge (\neg s(pc(t), a) \rightarrow k = 0) \wedge \ldots\\
&\rightarrow \mathit{inv}_0(pc, a, k)\ .
\end{aligned}$$

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

$$s(pc(t), a) \stackrel{\text{def}}{=} (pc(t) \geq 2)$$
$$\mathit{inv}_0(pc, a, k) \stackrel{\text{def}}{=} (k \leq a)\ .$$

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: $\mathit{init}(g, L)$, $\mathit{next}_{\mathrm{T}}(g, l, g', l')$, and $\mathit{safe}(g, L)$. Constraints $\mathit{init}(g, L)$ and $\mathit{safe}(g, L)$ 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 $\mathit{next}_{\mathrm{T}}(g, l, g', l')$ 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 $\mathit{next}_{\mathrm{T}}(g, l, g', l')$ 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* as follows.

$$\mathit{next}(g, L, g', L') \stackrel{\text{def}}{=} \exists t : \begin{pmatrix} \mathit{next}_{\mathrm{T}}(g, L(t), g', l') \wedge \\ L' = L[t \leftarrow l']) \end{pmatrix} \quad (1)$$

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 lockstep, 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 allows us to apply a standard proof rule for safety to describe a safe, inductive invariant $\mathit{inv}(g, L)$ 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.

$$\frac{\Delta \qquad \begin{array}{r} \mathit{Def}(k) = \#\{t \mid \varphi\} \\ \mathit{Def}(l) = \#\{t \mid \varphi'\} \end{array}}{((\forall t : \varphi \rightarrow \varphi') \rightarrow k \leq l) \wedge \Delta}$$

(a) Rule CARD$_{\leq}$.

$$\frac{\Delta \qquad \begin{array}{r} \mathit{Def}(k) = \#\{t \mid \varphi\} \\ \mathit{Def}(l) = \#\{t \mid \varphi'\} \end{array}}{(\begin{pmatrix} \forall t : \varphi \rightarrow \varphi' \wedge \\ (\exists t : \neg\varphi \wedge \varphi') \end{pmatrix} \rightarrow k < l) \wedge \Delta}$$

(b) Rule CARD$_{<}$.

$$\frac{\Delta \qquad \begin{array}{r} \mathit{Def}(k) = \#\{t \mid \varphi(t)\} \\ \mathit{Def}(l) = \#\{t \mid \varphi'(t)\} \\ g = f[j \leftarrow \_] \textit{ occurs in } \Delta \\ \varphi' = \varphi[g/f] \end{array}}{\begin{pmatrix} \mathbb{1}(\varphi'(j), \delta^+) \wedge \mathbb{1}(\varphi(j), \delta^-) \wedge \\ (l = k + \delta^+ - \delta^-) \end{pmatrix} \wedge \Delta}$$

(c) Rule CARD-UPD.

Figure 4: Rewriting rules for instantiating cardinality axioms.

5. Cardinality Axioms

Consider the combined theory of linear integer arithmetic and arrays, i.e., the theory of linear arithmetic extended with the interpreted functions $\cdot(\cdot)$ for array reads and $\cdot[\cdot \leftarrow \cdot]$ for array updates (see e.g. [Bradley et al. 2006] for more details). In order to extend this theory to incorporate cardinalities we distinguish variables of two sorts: variables of sort *thread identifier* that are used to index into arrays and variables of sort *integer* that are used to model data. Thread identifiers only support equality and inequality comparisons whereas integer variables support standard arithmetic operators. Let φ be a quantifier-free formula in that theory such that φ does not contain the update function. Then, for thread identifier t and integer variable k, we call an expression $\#\{t \mid \varphi\} = k$ a *cardinality constraint*. Let Ψ be a cardinality free formula. Then, in this paper, we consider formulas of the form

$$\#\{t \mid \varphi_1\} = k_1 \wedge \cdots \wedge \#\{t \mid \varphi_n\} = k_n \wedge \Psi\ .$$

In order to efficiently handle the cardinality free part Ψ, we assume that quantification is restricted in a way to ensure that a complete instantiation for the universal quantifiers can be efficiently computed (e.g., the formula falls into the array property fragment [Bradley et al. 2006] which admits

guarded universal statements). We note, however, that our axiomatization is sound in the case where Ψ is unrestricted. In order to avoid reasoning about infinite cardinalities, we assume that the set of all threads $\{t \mid true\}$ is of arbitrary but fixed size.

EXAMPLE 1. *The formulas*

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

and

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

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 Δ, our procedure first replaces all cardinality constraints by fresh variables, where the procedure maintains a bookkeeping function $Def(\cdot)$ that maps fresh variables to cardinalities. We assume that this function has a designated entry $Def(0) = \#\{t \mid false\}$ 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. Each rule specifies a re-writing of a formula Δ which strengthens the formula through a cardinality axiom. The right-hand side of the rule contains a number of preconditions that need to be satisfied in order for the rule to be applicable. Our axiomatization consists of three rules. We now describe the individual axioms in more detail.

- Rule CARD$_{\leq}$ instantiates a rule tracking *non-strict inequalities* between cardinalities.
- Rule CARD$_{<}$ instantiates a rule tracking *inequalities* between cardinalities.
- Rule 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 value at j was part of the set before the update, and incremented if the value at j is part of the set after the update. This is formalized through an indicator relation $\mathbb{1}$. For a constraint φ and variable k, we define $\mathbb{1}$ as follows.

$$\mathbb{1}(\varphi, k) \stackrel{\text{def}}{=} (\varphi \wedge k = 1) \vee (\neg\varphi \wedge k = 0)$$

The rule CARD-UPD can only be applied if defining formulas φ and φ' are equal, except for the use of array f and g respectively. Moreover, we require that arrays in φ and φ' may only be indexed by the variable bound in the set-comprehension. These conditions ensure that the only difference in the cardinality of both sets stems from the function update.

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

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

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

$$(\forall t : f(t) = 1) \wedge ((\forall t : f(t) \geq 2 \rightarrow false) \rightarrow k \leq 0) \wedge k \geq 1$$

which we simplify (for readability) into

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

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

$$f(t) = 1 \wedge (f(t) \geq 2 \vee k \leq 0) \wedge k \geq 1\ .$$

For the formula

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

instantiating axiom CARD-UPD *yields*

$$l = k + \delta^+ - \delta^- \wedge \mathbb{1}(g(j) = 2, \delta^+) \wedge \mathbb{1}(f(j) = 2, \delta^-) \wedge f(j) = 1 \wedge g = f[j \leftarrow 2] \wedge l \leq k$$

which simplifies to

$$l = k + 1 \wedge f(j) = 1 \wedge g = f[j \leftarrow 2] \wedge l \leq k$$

■

Soundness Our axioms are sound, which in turn underpins the soundness of #Π.

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

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

- CARD$_{\geq 0}$: cardinalities are always non-negative. That is for $k \in dom(Def)$ we have $k \geq 0$.
- CARD$_{\emptyset}$: if there is no element in a set, the cardinality of that set is zero. For $Def(k) = \#\{t \mid \varphi\}$ 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 $Def(k) = \#\{t \mid \varphi\}$ the following holds.
$$(\exists t : \varphi) \rightarrow k > 0$$

***Relative Completeness of* 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 CARD-UPD preserves difference bound constraints over cardinalities. That is, if a function update induces a relationship between the cardinalities of two sets and that relationship can be expressed as a difference bound constraint, then our update axiom captures that relationship.

THEOREM 2 (Relative completeness CARD-UPD). *Let Δ be an arbitrary formula in the combined theory of cardinality constraints, arrays and arithmetic. We let Ψ denote a formula containing the cardinality of two sets related through an update statement.*

$$\Psi \stackrel{\text{def}}{=} (\#\{t \mid \varphi\} = k) \wedge (\#\{t \mid \varphi'\} = l) \wedge g = f[j \leftarrow \_] \wedge \Delta$$

where $\varphi' = \varphi[g/f]$. Moreover, we require that arrays in φ and φ are indexed by only t. Let θ denote the same formula after the instantiation of the update axiom.

$$\theta \stackrel{\text{def}}{=} (l = k + \delta^+ - \delta^-) \wedge \mathbb{1}(\varphi', \delta^+) \wedge \mathbb{1}(\varphi, \delta^-) \wedge \Delta$$

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

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

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 Ψ such that Ψ is satisfiable formula in any theory that includes arithmetic, and for all difference constraints $\rho(k, l)$ and constants c, if*

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

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 θ hold, then we need to show $\rho(k, l)$. By case splitting over truth valuations for φ, and φ', 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 φ to direct array reads, all axiom instantiations fall into the array-property fragment, and we can therefore efficiently compute a complete instantiation for the 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 φ 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 $region(Q)$, which we define as follows.

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

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

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

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

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

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.*

$$\#\{t \mid f(t) = 1\} \geq \tfrac{2n}{3} \wedge \#\{t \mid g(t) = 1\} \geq \tfrac{2n}{3} \wedge \#\Omega = n \wedge \#\{t \mid f(t) = 1 \wedge g(t) = 1\} = 0$$

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

ing equations.

$$\begin{aligned}
\#\{t \mid a\} &= \#\{t \mid a \wedge \neg b\} + \#\{t \mid a \wedge b\} \\
\#\{t \mid b\} &= \#\{t \mid \neg a \wedge b\} + \#\{t \mid a \wedge b\} \\
\#\Omega &= \#\{t \mid a \wedge \neg b\} + \#\{t \mid \neg a \wedge b\} + \\
&\quad \#\{t \mid a \wedge b\} + \#\{t \mid \neg a \wedge \neg b\}
\end{aligned}$$

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

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

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

5.3 Limitations

Currently, the main limitation of our axiomatization with respect to the considered logic fragment is that we only support a limited form of case splits over thread identifiers. This support comes in the form of our update axiom, which encodes a case split in the following sense: an update $g = f[j \leftarrow v]$ can be translated into the following formula, which encodes a case split over the position j and all other positions.

$$g(j) = v \wedge (\forall t : t \neq j \rightarrow g(t) = f(t))$$

We now provide two examples of formulas which include case splits that induce relations between the cardinalities of sets that cannot be captured by our axiomatization.

EXAMPLE 3. *Consider the following formula describing arrays* f *and* g *which have the same value for all positions except for positions* i *and* j *where their values are swapped.*

$$\begin{aligned}
&i \neq j \wedge \\
&(\forall t : t \neq i \wedge t \neq j \rightarrow f(t) = g(t) \wedge g(t) = 1) \wedge \\
&f(i) = 1 \wedge g(i) = 2 \wedge \\
&f(j) = 2 \wedge g(j) = 1
\end{aligned}$$

Assuming the above formula, it holds that

$$\#\{t \mid f(t) = 1\} = \#\{t \mid g(t) = 1\}$$

however the formulas

$$\forall t : f(t) = 1 \rightarrow g(t) = 1$$

and

$$\forall t : g(t) = 1 \rightarrow f(t) = 1$$

do not hold, and hence rule CARD$_{\leq}$ *cannot be used to infer the above equality. Similarly, the following formula describes arrays* f *and* g *which have the same value on all positions except for* i *and* j*, however both positions are* 2 *for* f *and* 1 *for* g*.*

$$\begin{aligned}
&i \neq j \wedge \\
&(\forall t : t \neq i \wedge t \neq j \rightarrow f(t) = g(t) \wedge g(t) = 1) \wedge \\
&f(i) = 2 \wedge f(j) = 2 \wedge \\
&g(i) = 1 \wedge g(j) = 1 \,.
\end{aligned}$$

Assuming the above formula, it holds that $\#\{t \mid f(t) = 1\} \leq \#\{t \mid g(t) = 1\} + 2$*, however our axioms are not strong enough to derive this fact.* ■

While our limited support for case splits prevents our method from inferring certain relationships between cardinalities of sets, it allows us to avoid the potential blowup such a treatment would incur and hence ensures effectiveness of our method. We also note that none of the intricate reasoning patterns above occur in the examples in our evaluation.

6. The Method #Π

In this section, we describe our method #Π 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 [Hojjat et al. 2014; Bjørner et al. 2013].
- *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 [Grebenshchikov et al. 2012; Kahsai et al. 2015; Hoder and Bjørner 2012; Beyene et al. 2013]) 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 template* that fixes the number of cardinality expressions and universal quantifiers that are allowed to occur in the invariant. For an invariant *inv* with n quantifiers and m cardinality expressions, this defines an assertion $Shape(inv)$ of the following form, where inv_0 is an unknown quantifier-free assertion that relates cardinalities

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** $p \in dom(Shape)$ **and** $c \in C$ **do**
2: $c \leftarrow c[Shape(p)/p]$
3: $c \leftarrow \text{INSTQ}(Shape(p), c)$
4: **end**
5: $C \leftarrow \text{ELIMCARD}(C)$
    return SOLVE(C, Q)
  end
end

Figure 5: Algorithm #Π.

with program data, and $s_1, \ldots, s_m$ are unknown assertions defining the respective sets.

$$\forall q_1, \ldots, q_n : \quad \#\{t \mid s_1\} = k_1 \wedge \cdots \wedge \#\{t \mid s_m\} = k_m \wedge \mathit{inv}_0$$

EXAMPLE 4. *In the filter-lock example, we search for an invariant with 1 quantifier and 1 cardinality expression defining an expression* $Shape(\mathit{inv}) \stackrel{\text{def}}{=} \forall q : \#\{t \mid s\} \wedge \mathit{inv}_0$. ■

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 $Shape$. #Π returns a solution function Σ that maps each query to a constraint such that all clauses in C are valid if one substitutes each query by its solution. Function INSTQ(ψ, c) takes as input a quantified formula ψ and a clause c. It produces an instantiated clause as output. Function 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 INSTQ. It then invokes function ELIMCARD which instantiates cardinality axioms for the unknown assertions defining the sets. The resulting clauses are passed to a Horn solver, which returns a solution function.

EXAMPLE 5. *Consider again clause (a) from the example in Section 3 which we restate for convenience.*

$$\mathit{init}(a, pc) \rightarrow \mathit{inv}(a, pc)$$

Plugging in the shape template yields

$$\mathit{init}(a, pc) \rightarrow \#\{t \mid s(pc(t), a)\} = k \wedge \mathit{inv}_0(pc, a, k) \; .$$

Function INSTQ *leaves the clause unchanged as the shape template does not contain universal quantification. Procedure* ELIMCARD *subsequently instantiates cardinality axioms for the unknown assertion* $s(pc(t), a)$ *which yields the following clause where we expanded the definition for* $\mathit{init}(a, pc)$*. For simplicity we only instantiate the rule* $\text{CARD}_{\geq 0}$.

$$\begin{pmatrix} (\forall t : pc(t) = 1) \wedge a = 0 \wedge \\ (\forall t : \neg s(pc(t), a)) \rightarrow k = 0 \end{pmatrix} \rightarrow \mathit{inv}_0(pc, a, k)$$

Since the formula in the body of the clause falls into the array property fragment, ELIMCARD *can use the complete instantiation procedure form [Bradley et al. 2006] to instantiate quantifiers which yields the following clause in which we rewrote the implication in the bottom line as a disjunction.*

$$\begin{pmatrix} (pc(t) = 1) \wedge a = 0 \wedge \\ (s(pc(t), a)) \vee k = 0) \end{pmatrix} \rightarrow \mathit{inv}_0(pc, a, k)$$

This clause can be transformed into the two equivalent clauses

$$\begin{pmatrix} pc(t) = 1 \wedge a = 0 \wedge \\ s(pc(t), a) \end{pmatrix} \rightarrow \mathit{inv}_0(pc, a, k)$$

and

$$a = 0 \wedge pc(t) = 1 \wedge k = 0 \rightarrow \mathit{inv}_0(pc, a, k) \; .$$

Finally, the procedure passes the resulting clauses together with the clauses that result from running the algorithm on (b) and (c) to a Horn solver which returns the desired solution.

■

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 shape templates that specify the required number of quantifiers and set comprehension predicates as marked in the table.

Examples from [Farzan et al. 2014] The upper table shows result on examples taken from [Farzan et al. 2014]. 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

Program	Card	Property	Inferred cardinalities	Time
intro [Farzan et al. 2014]	✓	$(\exists t : pc(t) = 2) \rightarrow b < a$	$\#\{t \mid pc(t) = 2\}$	1.2s
bluetooth [Farzan et al. 2014]	✓	$(\exists t : pc(t) = 2) \rightarrow st = 0$	$\#\{t \mid pc(t) = 2\}$	1.6s
tree traverse [Farzan et al. 2014]	×	$leaves = nodes + 1$	–	4.2s
cache [Yongjian]	✓	$\#\{t \mid pc(t) = 3\} \leq 1$	$\#\{t \mid pc(t) \geq 3\}$	0.7s
garbage collection	✓	$\#\{t \mid 2 \leq pc(t) \leq 4\} \leq 1 \wedge m = 1$	$\#\{t \mid 2 \leq pc(t) \leq 4\}$	10.1s

Program	Property	Inferred cardinalities	Time
ticket lock [Farzan et al. 2014]	$\#\{t \mid pc(t) = 3\} \leq 1$	$\#\{t \mid m(t) \leq s \wedge pc(t) = 2\}$, $\#\{t \mid pc(t) = 3\}$, $\#\{t \mid m(t) = q\}$	20.9s
filter lock [Herlihy and Shavit 2008]	$\#\{t \mid lv(t) = n - 1\} \leq 1$	$\#\{t \mid lv(t) \geq q\}$	27.5s
one-third rule [Drăgoi et al. 2014; Charron-Bost and Schiper 2009]	see Section 2	$\#\{t \mid x(t) = x(q)\}$	0.8s

Figure 6: Applying #Π 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.

Program	Card	Correct	Property	Inferred cardinalities	Time	Time [Ganjei et al. 2015]
max [Ganjei et al. 2015]	✓	✓	$\exists t : pc(t) = 5 \rightarrow prev \leq max$	$\#\{t \mid pc(t) \leq 2\}$, $\#\{t \mid pc(t) \leq 3\}$, $\#\{t \mid pc(t) \geq 5\}$	4.2s	192s
max-nobar [Ganjei et al. 2015]	-	×	$\exists t : pc(t) = 5 \rightarrow prev \leq max$	-	7.2s	24s
reader/writer [Ganjei et al. 2015]	×	✓	$readcount > 0 \rightarrow writing = -1$	-	0.4s	38s
reader/writer-bug [Ganjei et al. 2015]	-	×	$readcount > 0 \rightarrow writing = -1$	-	0.5s	11s
parent/child [Ganjei et al. 2015]	✓	✓	$\exists t : pc(t) = 3 \rightarrow alloc = 1$	$\#\{t \mid 2 \leq pc(t) \leq 3\}$	1.2s	73s
parent/child-nobar [Ganjei et al. 2015]	-	×	$\exists t : pc(t) = 3 \rightarrow alloc = 1$	-	1.8s	3s
simp-bar [Ganjei et al. 2015]	✓	✓	$\exists t : pc(t) = 5 \rightarrow fl = 1$	$\#\{t \mid pc(t) \leq 3\}$ $\#\{t \mid pc(t) \leq 2\}$ $\#\{t \mid pc(t) = 5\}$	26.7s	93s
simp-nobar [Ganjei et al. 2015]	-	×	$\exists t : pc(t) = 5 \rightarrow fl = 1$	-	4.2s	13s
dyn-barrier [Ganjei et al. 2015]	✓	✓	$\#\{t \mid pc(t) \leq 2\} \leq 0$	$\#\{t \mid pc(t) \leq 2\}$, $\#\{t \mid pc(t) \geq 4\}$	1.3s	8s
dyn-barrier-nobar [Ganjei et al. 2015]	-	×	$\#\{t \mid pc(t) \leq 2\} \leq 0$	-	1.4s	3s
as-many [Ganjei et al. 2015]	✓	✓	$c_1 = c_2$	$\#\{t \mid pc(t) \geq 2\}$,	0.5s	62s
as-many-bug [Ganjei et al. 2015]	-	×	$c_1 = c_2$	-	0.7s	2s

Figure 7: Comparison to [Ganjei et al. 2015]. [Ganjei et al. 2015] 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. #Π outperforms [Ganjei et al. 2015] on all examples. We attribute this to the fact that #Π infers a suitable subset of relevant cardinalities.

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 [Yongjian], 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 [Abdulla et al. 2007] 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 [Farzan et al. 2014], 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.

[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.

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

Comparison with [Ganjei et al. 2015] Table 7 contains the results of a comparison with benchmarks taken from [Ganjei et al. 2015]. 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 [Ganjei et al. 2015] shows that our method outperforms [Ganjei et al. 2015] on all examples. We attribute this to the fact that #Π automatically discovers what to count and hence only tracks a small number of cardinalities, while cardinalities in [Ganjei et al. 2015] are tracked eagerly for each abstract state previously computed in a separate predicate abstraction phase. Our method benefits from templates which specify the number of cardinalities needed to complete the proof and which we assume are given as input for our method. We note however that for all examples, this number varies between one and three, and as a consequence it is easy to spawn a search space over it.

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 #Π to the methods from [Abdulla et al. 2007] and [Sanchez et al. 2012]. Table 9 summarises the results. We use shape templates that do not use cardinalities

```
   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*.

of sets and that use the number of quantifiers marked in column **Q**.

Benchmarks in [Abdulla et al. 2007] consist of a number of mutual-exclusion protocols that require invariants 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 [Abdulla et al. 2007] when using a solver over the reals and slightly faster when solving over integers. Examples from [Sanchez et al. 2012] 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 [Sanchez et al. 2012] 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.

Program	Q	Time		
		#Π	Real	Integer
Simplified Bakery	2	0.4s	0.8s	0.3s
Lamport's Bakery	2	0.5s	2.1s	2s
Bogus Bakery	2	0.6s	0.8s	11s
Ticket Mutex	2	0.5s	0.3s	1.6s

Program	Q	Time			
		#Π	I	P	O
barrier	1	0.4s	0.1s	0.1s	0.1s
central barrier	1	0.4s	0.1s	1.1s	6.2s
work stealing	1	0.5s	0.1s	0.1s	6.2s
dining philosophers	0	8.2s	0.1s	6.3s	20s
robot 2x2	2	2.8s	0.2s	5.8s	1m45s
robot 2x3	2	16.1s	0.5s	16s	5m20s
robot 3x3	2	34.0s	0.9s	52s	19m28s
robot 4x4	2	TO	3.2s	5m3s	TO

Figure 9: Cardinality-free reasoning: Results of comparing #Π to [Abdulla et al. 2007] and [Sanchez et al. 2012]. 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 [Kuncak et al. 2005] and generalized to multi-sets and fractional collections in [Piskac and Kuncak 2008a,b] and direct and inverse function/relation images in [Yessenov et al. 2010]. 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.

[v. Gleissenthall et al. 2015] 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.

[Fredrikson and Jha 2014] introduces the problem of *model-counting satisfiability* which, given an SMT formula that contains a number of parameters, requires finding assignments to the parameters such that the resulting formula satisfies a given cardinality constraint. [Fredrikson and Jha 2014] is not directly applicable to our setting as it focuses on *checking* correctness arguments rather than *synthesising* them. Moreover [Fredrikson and Jha 2014] requires a model counting procedure. To the best of our knowledge, no such procedure exists in a theory containing uninterpreted functions.

Dragǒi et al. propose a logic that contains cardinality constraints over uninterpreted functions as well as limited quantifier alternation in [Drăgoi et al. 2014]. This logic is geared towards the verification of consensus protocols such as Paxos [Lamport 1998] in the heard-of model [Charron-Bost and Schiper 2009] which allows for benign (communication) faults. While the logic is similar in spirit to our approach, [Drăgoi et al. 2014] focuses on satisfiability checking in an expressive subset of first order logic with the primary intent of *checking* inductive correctness arguments, whereas our focus lies on *synthesizing* such arguments automatically (in a more restricted setting).

[Ganjei et al. 2015] 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. One important implication of this restriction is that [Ganjei et al. 2015] is not able to discover proofs that require relating locals of more than one thread. Algorithmically, our approach tightly integrates the discovery of what to count and the construction of the invariant that uses the count. This is in contrast to [Ganjei et al. 2015] where first a finite state abstraction is created, then, a counter is given to each abstract state and finally invariants over counters are discovered.

The abstract interpretation based approach presented in [Gulwani et al. 2009] 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.

While the focus of this paper lies on automating the discovery of invariants for parametrized systems that contain cardinality constraints, we think that studying the complexity of underlying logic fragments is an important research direction. Recently, [Alberti et al. 2016] proposed an extension of the flat-array segment with cardinality constraints and showed its decidability. As the algorithm for checking satisfiability of the logic can lead to a substantial blow-up due to a form of Venn decomposition, [Alberti et al. 2016] identifies a sub-fragment for which checking can be efficiently automated, and applies this method to check invariants for safety properties of distributed algorithms. The restriction of

this sub-fragment coincides with our restriction made in Remark 1, i.e., that the variable bound in the set comprehension can only be used to directly index into arrays. In our setting, this restriction ensures that all axiom instances fall into the array property fragment.

Quantitative Verification of Parametric Systems A classic example of the use of quantitative abstractions for parametric system is [Pnueli et al. 2002a], where a number of bounded auxiliary counters for predefined sets of states are used to prove liveness of parametric protocols. The CIRC extension [Henzinger et al. 2004a] of Blast [Henzinger et al. 2002, 2004b] shows how auxiliary counters can be inferred under predicate abstraction. [Basler et al. 2009b] 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 [Farzan et al. 2014] 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 [Farzan et al. 2014] 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 [Balaban et al. 2005; Fang et al. 2006] relies on small instantiation to generate candidates for universally quantified array invariants and proposes fragments where checking these candidates can be done effectively, even in the presence of complex communication topologies [Balaban et al. 2006]. Our approach computes quantifier instantiation as a part of the inference process. In [Kaiser et al. 2014] 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 expressing properties such as: "variable m of this thread is smaller than the variable m of all other threads", which enables verifying the protocols like 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. [Sanchez et al. 2012] 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. [Abdulla et al. 2007] introduces a formalism that allows to express global conditions which relate local variables of different threads, and uses backward reachability to verify safety properties. [Farzan et al. 2015] explores the notion of *proof spaces*, in which a finite number of Hoare triples is combined through a fixed set of rules in order to prove properties about parametrized systems. Both [Farzan et al. 2014] and [Farzan et al. 2015] are based on the language-theoretic approach to program correctness introduced in [Heizmann et al. 2009]. The use of data flow graphs in [Farzan and Kincaid 2012] 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.

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 scarce. This paper presented #Π, a method 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 #Π yielded an effective tool that is capable of verifying intricate parameterized systems using cardinality arguments. At the same time #Π 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 #Π could provide some information, following [Gulwani et al. 2008], but this is currently not explored.)
- We do not investigate the effectiveness of #Π for modular reasoning in the presence of procedures. (Targeting the case when procedures coincide with transactions [Qadeer et al. 2004] appears to be a promising direction to consider.)

Acknowledgments

Klaus v. Gleissenthall was supported through a Microsoft Research scholarship. We would like to thank Matthew Parkinson, our shepherd Matt Fredrikson, and the anonymous reviewers for their comments which helped significantly improve the presentation of this paper.

References

P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV*, 2007.

F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. An extension of lazy abstraction with interpolation for programs with arrays. *FMSD*, 45(1), 2014a.

F. Alberti, S. Ghilardi, and N. Sharygina. A framework for the verification of parameterized infinite-state systems. In *CILC*, 2014b.

F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for flat array properties. *J. Autom. Reasoning*, 54(4), 2015.

F. Alberti, S. Ghilardi, and E. Pagani. Counting constraints in flat array fragments. In *IJCAR*, 2016.

I. Balaban, Y. Fang, A. Pnueli, and L. D. Zuck. IIV: an invisible invariant verifier. In *CAV*, 2005.

I. Balaban, A. Pnueli, and L. D. Zuck. Invisible safety of distributed protocols. In *ICALP*, 2006.

G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. Symbolic counter abstraction for concurrent software. In *CAV*, 2009a.

G. Basler, M. Mazzucchi, T. Wahl, and D. Kroening. Symbolic counter abstraction for concurrent software. In *CAV*, 2009b.

T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In *CAV*, 2013.

N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified horn clauses. In *SAS*, 2013.

A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays. In *VMCAI*, 2006.

M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

B. Charron-Bost and A. Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 2009.

C. Drăgoi, T. A. Henzinger, H. Veith, J. Widder, and D. Zufferey. A logic-based framework for verifying consensus algorithms. In *VMCAI*, 2014.

Y. Fang, N. Piterman, A. Pnueli, and L. D. Zuck. Liveness with invisible ranking. *STTT*, 8(3), 2006.

A. Farzan and Z. Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *POPL*, 2012.

A. Farzan, Z. Kincaid, and A. Podelski. Proofs that count. In *POPL*, 2014.

A. Farzan, Z. Kincaid, and A. Podelski. Proof spaces for unbounded parallelism. In *POPL*, 2015.

M. Fredrikson and S. Jha. Satisfiability modulo counting: A new approach for analyzing privacy properties. In *LICS*, 2014.

Z. Ganjei, A. Rezine, P. Eles, and Z. Peng. Abstracting and counting synchronizing processes. In *VMCAI*, 2015.

S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, 2008.

S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *POPL*. ACM, 2009.

C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, , and B. Zill. IronFleet: Proving practical distributed systems correct. In *SOSP*, 2015a.

C. Hawblitzel, E. Petrank, S. Qadeer, and S. Tasiran. Automated and modular refinement reasoning for concurrent programs. In *CAV*, 2015b.

M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of trace abstraction. In *SAS*, 2009.

T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, 2004a.

T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004b.

M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.

H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A verification toolkit for numerical transition systems - tool paper. In *FM*, 2012.

H. Hojjat, P. Rümmer, P. Subotic, and W. Yi. Horn clauses for communicating timed systems. In *HCVS*, 2014.

P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX*, 2010.

T. Kahsai, J. A. Navas, A. Gurfinkel, and A. Komuravelli. The SeaHorn verification framework. In *CAV*, 2015.

A. Kaiser, D. Kroening, , and T. Wahl. Lost in abstraction: Monotonicity in multi-threaded programs. In *CONCUR*, 2014.

R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong. Zyzzyva: speculative Byzantine fault tolerance. In *SOSP*, 2007.

D. Kroening and M. Lewis. Second-order SAT solving using program synthesis. *CoRR*, abs/1409.4925, 2014.

V. Kuncak, H. H. Nguyen, and M. C. Rinard. An algorithm for deciding BAPA: Boolean Algebra with Presburger Arithmetic. In *CADE*, 2005.

L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998.

L. Lamport. Mechanically checked safety proof of a byzantine Paxos algorithm, 2015. `http://research.microsoft.com/users/lamport/tla/byzpaxos.html`.

Concurrent Garbage Collection. .NET Framework 4.6 and 4.5. Microsoft, 2015. `https://msdn.microsoft.com/en-us/library/ee787088(v=vs.110).aspx#concurrent_garbage_collection`.

D. Monniaux and F. Alberti. A simple abstraction of arrays and maps by program translation. *SAS*, 2015.

C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon web services uses formal methods. *Commun. ACM*, 2015.

D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, 2014.

R. Piskac and V. Kuncak. Fractional collections with cardinality bounds, and mixed linear arithmetic with stars. In *CSL*, 2008a.

R. Piskac and V. Kuncak. Decision procedures for multisets with cardinality constraints. In *VMCAI*, 2008b.

A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, infty)-counter abstraction. In *CAV*, 2002a.

A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, infty)-counter abstraction. In *CAV*, 2002b.

S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *POPL*, 2004.

A. Sanchez, S. Sankaranarayanan, C. Sànchez, and B.-Y. E. Chang. Invariant generation for parametrized systems using self-reflection. In *SAS*, 2012.

I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, 2015.

T. Terauchi and H. Unno. Relaxed stratification: A new approach to practical complete predicate refinement. In *ESP*, 2015.

H. Unno and T. Terauchi. Inferring simple solutions to recursion-free horn clauses via sampling. In *TACAS*, 2015.

K. v. Gleissenthall, B. Köpf, and A. Rybalchenko. Symbolic polytopes for quantitative interpolation and verification. In *CAV*, 2015.

K. Yessenov, R. Piskac, and V. Kuncak. Collections, cardinalities, and relations. In *VMCAI*, 2010.

L. Yongjian. A novel approach to the parameterized verification of cache coherence protocols. In *Tech Report.* `http://lcs.ios.ac.cn/~lyj238/papers/techReportCache.pdf`.

P. Zave. How to make Chord correct (using a stable base). *CoRR*, abs/1502.06461, 2015.