Technischen Universität München

Chair for Foundations of Software Reliability and Theoretical Computer Science

Cardinalities in Software Verification

Klaus Ulrich Kurt Freiherr von Gleissenthall

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Tobias Nipkow, Ph.D.

Prüfer der Dissertation: 1. Prof. Dr.-Ing. Andrey Rybalchenko

2. Prof. Dr. Andreas Podelski

Die Dissertation wurde am 24.02.2016 bei der Technischen Universität München eingereicht und durch die Fakultaet für Informatik am 26.07.2016 angenommen.

Abstract

Many verification tasks require reasoning about cardinalities. For example, verifying the correctness of consensus protocols requires tracking the cardinality of the set of distributed nodes that have agreed on a common value. Similarly, proving quantitative bounds on information leakage requires bounding the number of observations an attacker can make about a secret.

Unfortunately, today's technology does not support the verification of programs that require reasoning about cardinalities very well. In this thesis, we present two methods #Horn and #Π [1] for this task. #Horn introduces cardinalities as *properties* that one can verify about a program, while #Π introduces cardinalities into *proofs*.

We have implemented our techniques and demonstrated their practicality on a set of benchmarks from various applications including the ones mentioned above.

[1] Pronounced as "sharpie".

Zusammenfassung

Für viele Verifikationsaufgaben ist es unerlässlich die Anzahl der Elemente in einer Menge (das heißt deren Kardinalität) erfassen zu können. So erfordert beispielsweise die Verifikation vieler Netzwerk Algorithmen die Erfassung der Anzahl an teilnehmenden Netzwerk Knoten die eine bestimmte Eigenschaft teilen. Man stößt auf ein ähnliches Problem wenn man die Menge an (sicherheitsrelevanten) Informationen beschränken möchte die ein Programm über Seitenkanäle preisgibt. In diesem Fall muss eine Verifikationsmethode die Anzahl der Beobachtungen beschränken die ein Angreifer über das Programm machen kann.

Leider wird diese Art der quantitativer Verifikation von den derzeitig existierenden Methoden nicht unterstützt. Aus diesem Grund entwickelt diese Dissertation zwei Methoden #Horn und #Π für diese Aufgabe. #Horn führt Kardinalitäten von Mengen als zu verifizierende Eigenschaft des Programms ein während #Π Kardinalitäten zusätzlich als Bausteine für den Korrektheitsbeweis nutzt.

Wir haben beide Techniken implementiert und demonstrieren ihre Anwendbarkeit auf eine Reihe von Problemen welche die oben genannten Anwendungen mit einschließt.

Acknowledgements

First and foremost, I would like to thank my advisor Andrey Rybalchenko for introducing me to the exciting area of program verification, teaching me the craft of research (and that simplicity is key), and his constant advice and encouragement.

I would like to thank Boris Köpf who acted as my second, unofficial adviser. Thank you for the many discussions from which I learned so many valuable lesson on how to do research and how to clearly present the findings.

I would like to thank Microsoft Research for financial support, and hosting me in their Cambridge lab for the biggest part of my doctoral studies.

I am in debt to Christian Osendorfer whose enthusiasm and brilliant teaching in his machine learning course got me interested in doing research. Jörg Kreiker's course on complexity theory first got me interested in theoretical computer science. Thank you for being such an inspiring teacher.

I'm glad to have had many great collegues in Munich, not least the members of our group Corneliu Popeea, Tewodros Beyene and Ruslan Lesdema Garza. I'm happy for all the brilliant people I met in Cambridge. Thanks to Nik Sultana, Heidy Khlaaf, Alexey Bakhirkin, Steven Woodhouse, Kathryn Atwell, Sarah Winkler, Hilda Mujcic, Oliver Crawford, Benjamin Choo and Nuno Lopes for making my stay fun.

I would like to thank Daniele Varacca and Jean Krivine for inviting me to LIAFA.

Last I would like to thanks my family and friends for their support. Thanks to all my Munich friends in particular Jan, Maddin and Basti. Finally I owe the biggest dept to my parents without whom none of this would have been possible.

Contents

Equations and Definitions

Chapter 1

Introduction

Many verification tasks require reasoning about cardinalities of sets.

Consider, for example, the task of providing correctness guarantees for consensus protocols. Such protocols form a backbone of the Internet as they provide data-consistency when computations are distributed across several locations. Consistency is provided through a method which supplies a communication primitive that ensures the consistent broadcast of a single piece of information across the network. This single broadcast operation is then used as a building block for larger applications. More technically, a consensus protocol takes as input a set of candidate values proposed by the distributed nodes and ensures that after executing the protocol all nodes agree on one of the values that were proposed initially. In such protocols, a node often takes decisions depending on the cardinality of the set of all nodes that already agree on its current candidate value. Thus, to prove correctness of the protocol, a verification method needs to track the cardinality of the above set.

Similarly, bounding the information leakage of a program requires bounding the cardinality of the set of all observations an attacker can make about the secret input values that the program processes. For example, if a (deterministic) program produces the same output regardless of which secret input is processed, the program's output does not allow to draw any conclusion about which secret was used. If, however, the program produces a large number of different outputs, this potentially reveals parts of the secret. Hence, bounding information leakage of programs requires verification technology that can track the cardinality of the set of all observations.

Unfortunately, tracking cardinalities is not well supported by today's automated verification technology.

Verifiers usually establish the correctness of a program by constructing a program *invariant*. An invariant is an assertion that holds on all reachable states of the program. Thus, if an invariant implies that no error states can be reached, one can conclude that the program is correct. Constructing such invariants by hand is tedious. Hence automation is desirable and many methods for invariant construction have been proposed (e.g., Astree [20], Blast [51], CPAChecker [16], UFO [2]). These methods are however purely *qualitative* and hence cannot be applied for reasoning about cardinalities. For example, these methods can track wether or not *there exists* a thread at a given program location or wether *all* threads are at that location, but they cannot track the *number* of threads at that location. Similarly, these methods can track wether or not a program leaks *some* information or *no* information at all, but they cannot quantify *how much* information it leaks.

Cardinality aware invariant generation In this thesis, we contribute two invariant generation methods for verification tasks that involve cardinalities (including the ones described above). The first method #Horn introduces cardinality constraints as *properties* that one would like to prove about the program; the second method #Π introduces cardinalities as part of the *proof*. We now describe the two methods in more detail.

- #Horn proves bounds on the cardinality of (some subset of) the reachable states. This is a refinement of the classic (safety) verification setting: instead of proving that *no* (error) states are reachable, one bounds the *number* of reachable states. For this task, #Horn constructs program invariants (in the theory of linear integer arithmetic) that satisfy constraints on the cardinality of the set of points that the invariant represents. The need to construct such cardinality constrained invariants naturally arises in the context of bounding resource usage and information leakage.

  For an example of a verification task where a resource bound is encoded through a cardinality constraint, consider a program that manipulates elements in a map that is indexed by a tuple of integer values, and assume that the number of elements that are accessed depends on some parameter n, e.g., because a loop is iterated n times. Then, the task of proving that the program accesses less than n^2 *map-elements* can be reduced to bounding the number of *map-indices* that were accessed by the program. This requires constructing a relational invariant $\mathit{reach}(\mathit{ind}, n)$ which is an assertion over the tuple of indices ind and parameter n, such that for each value of n, the set

  $$\{\mathit{ind} \mid \mathit{reach}(\mathit{ind}, n)\}$$

  over-approximates the set of indices at which the map is accessed for that choice of parameter, and which satisfies the following cardinality constraint stating that

for any choice of parameter, the number of indices that were accessed is bounded by n^2.

$$\forall n : n \geq 0 \rightarrow \#\{ind \mid reach(ind, n)\} \leq n^2$$

- #Π constructs invariants in the combined theory of linear arithmetic and uninterpreted functions where the invariants may refer to the cardinality of sets. Such invariants are often required in the verification of distributed systems, where uninterpreted functions serve to encode the local state of individual nodes or threads.

  Consider for example a simple multi-threaded program that accesses some shared resource, and assume for simplicity that a thread accesses the shared resource only if its program counter is set to location 2. Furthermore assume that the *number* of threads this program is supposed to be executed on cannot be determined a priori, and we want to prove that *for any number of threads* executing the program, the shared resource can only be accessed by *one thread* at a time.

  We can model this situation by representing local variables as functions from thread identifier to local state. For example, we let $pc(t)$ denote the program counter of a thread t. Then, we can use universal quantification over thread identifiers to make assertions over *all* threads (independently of how many threads *all* threads actually are), and existential quantification to make assertions about the actions that *some* thread performs without actually naming that thread.

  By modelling the program this way, we can use #Π to construct an invariant that proves that the shared resource is accessed correctly. For example, the invariant

  $$\#\{t \mid pc(t) = 2\} \leq 1$$

  states that the number of threads that access the shared resource at any given time is bounded by one. This implies that only one thread at a time can access the shared resource and hence proves correctness of the program.

Contributions By presenting #Horn and #Π, this thesis advances the state of the art in (quantitative) verification in the following ways: *conceptually*, it promotes cardinalities as first-class citizens in program verification; *practically*, it demonstrates that cardinality-aware verification can be performed efficiently, and *technically* it identifies the core insights needed to enable the tracking of cardinalities.

We now discuss these three contributions in more detail.

Cardinalities as first-class citizens The two invariant generation methods presented in this thesis introduce cardinalities (of interpreted sets) as first-class citizens

into the verification process. In the past, such a direct treatment has often been avoided as cardinality reasoning is considered to be notoriously difficult. As an alternative to a direct treatment, cardinality reasoning was *approximated*, e.g. by instrumenting the program with auxiliary counters. Such an (over-)approximation of cardinality comes, however, with a major drawback: if the chosen approximation happens to be too imprecise to track the desired property, there is no principled way of refining the approximation other than manually constructing a new one. In contrast, our methods track cardinality *precisely* and ensure effectiveness by harnessing existing verification technology to find suitable abstractions which approximate *the program* that needs to be verified rather than *the cardinalities* that need to be tracked.

Implementation and Evaluation We demonstrate the practicality of our methods, by implementing and evaluating both #Horn and #Π.

To evaluate #Horn, we applied our implementation on benchmarks from resource bound verification. For these benchmarks, our general purpose cardinality solver is as fast as specialized resource-bounds verifiers. This indicates that tracking cardinalities precisely does not incur a performance penalty. To demonstrate its ability to treat examples from other application domains, we show that #Horn can be used to automatically synthesise padding loops which ensure that the information leaked by a program is below a user-specified threshold. Performing just the amount of padding needed in order to maintain given bounds on information leakage allows one to balance the amount of unnecessary work with guarantees on leakage. We are not aware of any other automated method for this task.

To evaluate #Π, we tested our implementation on a number of benchmarks including mutual exclusion, consensus and cache-coherence protocols as well as a simple model of a mark-and-sweep garbage collector. Except for one benchmark, all of these examples were automatically verified for the first time. Since #Π can also be used for constructing invariants that *do not* refer to cardinalities, we have evaluated it on a number of benchmarks from the parametrized systems literature. This evaluation shows that for these benchmarks, #Π is as fast as and sometimes faster than existing methods.

Underlying Techniques For each method, we identify one main technical insight that enables our treatment of cardinalities.

For #Horn, the main technical insight is that we can use *existing mathematical theory* to (automatically) learn from failed verification attempts.

#Horn is built on an abstraction/refinement scheme [28]. That is, #Horn iteratively constructs an invariant by considering abstract versions of the program that we want to verify and trying to prove these abstract versions correct. If the abstract version of the program can be proved correct, then, by the fact that the abstraction is built in a way that ensures that the abstraction will only *add* new behaviour and never *remove* any existing behaviour from the original program, one can conclude that the original program is correct.

In the case where the abstract program *cannot* be proved correct, this can be due to one of two causes. Either, the original program is indeed incorrect, or the failure to prove the program results from the additional behaviour that was introduced by the abstraction. In the second case, #Horn recovers information from the failed verification attempt in order to refine the abstraction. This requires finding an *interpolant* which is a formula specifying which additional behaviours to remove from the abstraction in order to avoid re-discovery of the same spurious error. Technically an interpolant in our setting is a linear integer arithmetic formula such that: a) the set of states represented by that formula includes a given subset of reachable program states, and b) the cardinality of the set satisfies a specified bound. In mathematical terms, the problem of finding such an interpolant translates into the problem of synthesizing a *polytope* that satisfies a constraint on the number of *integer points* it contains. We exploit this insight by conceptually reversing the theory of counting integer points in polytopes [11, 22]. That is, instead of using the theory to *count* the number of integer points in a given polytope (its traditional use case), we use it to *synthesise* a polytope that satisfies a given count. This insight forms the basis for our interpolation method #Itp, which we present in the next chapter.

For #Π, the main technical insight is that properties of distributed systems often exhibit a *locality property* which ensures that whenever a distributed node updates its state, we just have to track whether this node moves in or out of a given set, i.e., we only have to reason about *one node at time*.

As cardinality reasoning is not supported by standard verification technology, our method needs to supply a semantics for cardinality reasoning. We provide this semantics in the form of an *axiomatization* of cardinality. Our method employs this axiomatization in order to reduce reasoning about cardinalities to reasoning about the *cardinality-free* part of the underlying theory (i.e., linear arithmetic and uninterpreted functions), allowing us to capitalize on the maturity of existing verification methods for this setting. Our axiomatization is based on the following insight: to reason about sets which just refer to the *local state* of threads, it suffices to track how the state change of the node *currently performing an update* evolves the cardinality of the set, i.e. whether the given

node moves into the set, moves out of the set or stays in the set. This gives us a strong locality property which allows very efficient verification. In our experiments, seven out of the nine sets that our method synthesises fall into this fragment. In order to deal with the case where sets may also refer to *global* state, we extend our axiomatization with a form of Venn-region decomposition, which case-splits over overlapping sets.

Outline and Sources This thesis consists of the following parts.

- Chapter 2, presents #Horn, our method for constructing cardinality constrained invariants in the theory of linear arithmetic. The main focus of this chapter lies on the interpolation method #Itp. This chapter is based on [88].
- Chapter 3, presents #Π, our method for constructing invariants in the combined theory of linear arithmetic, uninterpreted functions and cardinality constraints. This chapter is based on [19].
- Chapter 4 concludes.

Chapter 2

Interpolation with cardinality constraints

2.1 Introduction

Proving quantitative properties of programs often leads to verification conditions that involve cardinalities of sets and relations over program states. For example, determining the memory requirements for memoization reduces to bounding the cardinality of the set of argument values passed to a function, and bounding information leaks of a program reduces to bounding the cardinality of the set of observations an attacker can make.

A number of recent advances for discharging verification conditions with cardinalities consider extensions of logical theories with cardinality constraints, such as set algebra and its generalizations [65, 77, 78], linear arithmetic [33, 91], constraints over strings [69], as well as general SMT based settings [41]. At their core, these approaches operate by *checking* whether a cardinality bound holds for a given formula that describes a set of values. However, they cannot *synthesize* formulas that satisfy given cardinality constraints. As a consequence, the problem of automatically inferring cardinality-constrained inductive invariants remains an open challenge.

In this chapter, we present an approach for synthesizing linear arithmetic formulas that satisfy given cardinality constraints. Our approach relies on the theory of counting integer points in polytopes, however, instead of computing the cardinality of a given polytope (the typical use case of this theory) our approach synthesizes a polytope for a given cardinality constraint. Our synthesizer internally organizes the search space in terms of *symbolic polytopes*. Such polytopes are represented using symbolic vertices and hyperplanes, together with certain well-formedness constraints. We derive an expression

for the number of points in the polytope in terms of this symbolic representation, which leads to a set of constraints that at the same time represent the shape *and* the cardinality of the polytope. For this, we restrict our attention to the class of *unimodular* polytopes. Unimodularity can be concisely described using constraints and provides an effective means for reducing the search space while being sufficiently expressive. We then resort to efficient SMT solvers specifically tuned to deal with the resulting kind of non-linear constraints, e.g., Z3 [34]. We cast our approach in terms of an algorithm #ITP$_{\text{LIA}}$ for cardinality constrained interpolation, that is, #ITP$_{\text{LIA}}$ generates formulas that satisfy cardinality constraints along with implication constraints.

We put cardinality-constrained interpolation to work within an automatic verification method #HORN for inferring cardinality-basecd inductive program properties, based on abstraction and its counterexample-guided refinement. Specifically, #HORN is a solver for recursive Horn clauses with cardinality constraints. We rely on Horn clauses as basis because they serve as a language for describing verification conditions for a wide range of programs, including those with procedures and multiple threads [18, 43, 82]. Adding recursion enables representing verification conditions that rely on inductive reasoning, such as loop invariants or procedure summaries. By offering support for cardinalities directly in the language in which we express verification conditions, our solver can effectively leverage the interplay between the qualitative and quantitative (cardinality) aspects of the constraints to be solved.

We implemented #ITP$_{\text{LIA}}$ and #HORN and applied them to analyze a collection of examples that show

- how a variety of cardinality-based properties (namely, bounds on information leaks, memory usage, and execution time) and different program classes (namely, while programs and programs with procedures) can be expressed and analyzed in a uniform manner.
- that our approach can establish resource bounds on examples from the recent literature at competitive performance and precision, and that it can handle examples whose precise analysis is out of scope of existing approaches.
- that our approach can be used for synthesizing a padding-based countermeasure against timing side channels, for a given bound on tolerable leakage.

In summary, we contribute and put to work a synthesis method for polytopes that satisfy cardinality constraints, based on symbolic integer point counting algorithms.

2.2 Example: proving effectiveness of memoization

We consider a procedure `mcm` for *Matrix chain multiplication* [30] that recursively computes the cost of multiplying matrices $M_0, \ldots, M_n$ with optimal bracketing. `mcm(i, j)` returns the number of operations required for multiplying the subsequence $M_i, \ldots, M_j$, and `c(k)` returns the number of operations required for multiplying matrices M_k and M_{k+1}.

```
int mcm(int i, int j) {
 if (i == j) return 0;
 int minCost = infty;
 for (int k=i; k <= j-1; k++) {
  int v = mcm(i, k)+mcm(k+1, j)+c(k);
  if (v < minCost) minCost = v;
 }
 return minCost;
}
int main(n){
  mcm(0, n);
}
```

Even though the number of recursive function calls is exponential in n, `mcm` can be turned into an efficient algorithm by applying memoization. The amount of memory required to store results of recursive calls is bounded by

$$\frac{(n+1)\cdot(n+2)}{2}\,,$$

as `mcm` is only called with ordered pairs of arguments.

Proving such a bound requires reasoning about recursive procedure calls as well as tracking dependencies between variables i and j, i.e., estimating the range of each variable in isolation and combining the estimates is not precise enough.

When using #Horn, we first set up recursive Horn constraints on an assertion

$$\mathit{args}(i, j, n)$$

that contains all triples (i, j, n) such that calling `main`(n) leads to a recursive call `mcm`(i, j), following [43]. Then, #Horn solves these constraints using a procedure based on counterexample-guided abstraction refinement. As an intermediate step, #Horn deals with an interpolation query that requires finding a polytope φ_{args} over

i, j and n such that

$$n \geq 2 \wedge (i = 0 \wedge j = n \vee i = 1 \wedge j = 1) \rightarrow \varphi_{args} \quad (1)$$
$$n \geq 0 \rightarrow \#\{(i, j) \mid \varphi_{args}\} \leq \tfrac{(n+1)\cdot(n+2)}{2} \ . \quad (2)$$

Constraint (1) requires that φ_{args} contains triples obtained by symbolically executing `mcm`, a typical interpolation query, while (2) ensures that φ_{args} satisfies the bound by referring to the cardinality of φ_{args} through an application of cardinality operator $\#\cdot$.

Given (1) and (2), $\#\text{ITP}_{\text{LIA}}$ computes the solution

$$\varphi_{args} = (0 \leq i \leq 1 \wedge i \leq j \leq n \wedge n \geq 2) \ .$$

The cardinality of

$$\{(i, j) \mid \varphi_{args}\}$$

is $2n + 1$, hence φ_{args} satisfies the above bound. #HORN uses this solution to refine its abstraction. In particular, it starts using the predicate

$$i \leq j \ ,$$

which is crucial for tracking that `mcm` is only called on ordered pairs.

2.3 Counting integer points in polytopes

In this section, we first revisit the theory of counting integer points in polytopes. We then discuss the derivation of expressions for the number of integer points in unimodular polytopes with symbolic vertices and hyperplanes.

Preliminaries A *polytope* is the convex hull of a finite set of points. Each polytope P is defined by the set of its *vertices*, where a point in P is called a vertex if and only if it cannot be described as the convex combination of two other points in P. In this thesis, we restrict our attention to polytopes with integer vertices.

Let $g_1, \ldots, g_d \in \mathbb{R}^d$ be vectors in d-dimensional space. A *cone* with generators $g_1, \ldots, g_d$ is the set of all positive linear combinations of its generators. A cone is *unimodular* if and only if the absolute value of the determinant of the matrix $(g_1 \ \ldots \ g_d)$ is equal to one. [1]

[1] Section 2.4 provides examples and equivalent defintions of unimodularity. For more details, see e.g. [11, 12, 33].

The *vertex cone* of a polytope P at vertex v is the smallest cone that originates from v and that includes P; we denote its generators by $g_{v1}, \ldots, g_{vd}$. Finally, a polytope P is unimodular if all its vertex cones are unimodular.

Generating functions The integer points contained in a set $S \subseteq \mathbb{R}^d$ in can be represented in terms of a *generating function* $f(S, x)$ which is a sum of monomials, one per integer point in S, defined as follows

$$f(S, x) = \sum_{m \in S \cap \mathbb{Z}^d} x^m, \tag{2.1}$$

where for $m = (m_1, \ldots, m_d)$ we define

$$x^m = x_1^{m_1} \cdot \ldots \cdot x_d^{m_d} \ .$$

This generating function is a Laurent series, i.e. its terms may have negative degree. Note that, for finite S, the value of $f(S, x)$ at $x = (1, \ldots, 1)$, corresponds to the number of integer points in S.

Rational function representation Generating functions are a powerful tool for counting integer points in polytopes. This is due to two key results: First, Brion's theorem [22] allows to decompose the generating function of a polytope into the sum of the generating functions of its vertex cones. Second, the generating function of *unimodular* vertex cones can be represented through an equivalent yet short rational function. This rational function representation relies on a generalization of the equivalence

$$\frac{1}{1-x} = (1 + x + x^2 + x^3 + \ldots),$$

which provides a concise representation of the set $\{0, 1, 2, 3, \ldots\}$. [2]

This yields the following rational function representation for a unimodular polytope P with vertices $\mathcal{V}$:

$$r(P, x) = \sum_{v \in \mathcal{V}} \frac{x^v}{(1 - x^{g_{v1}}) \cdots (1 - x^{g_{vd}})} \tag{2.2}$$

Here, each summand represents the generating function of the vertex cone at v with generators $g_{v1}, \ldots, g_{vd}$. Rational function representations for arbitrary polytopes can be

[2] We provide more details on this derivation in Section 2.4.

obtained through Barvinok's algorithm [11] that decomposes arbitrary vertex cones into unimodular cones.

Generating function evaluation Since $x = (1, \ldots, 1)$ is a singularity of $r(P, x)$, computing the number of points in the polytope by direct evaluation leads to a division by zero. This can be avoided by performing a Laurent series expansion of $r(P, x)$ around $x = (1, \ldots, 1)$, however, the expansion requires a reduction of $r(P, x)$ from a multivariate polynomial over $(x_1, \ldots, x_d)$ to a univariate polynomial over y, see [33]. The reduction is done by finding a vector $\mu = (\mu_1, \ldots, \mu_d)$ with

$$\mu \cdot g \neq 0 \ , \tag{2.3}$$

for all generators g of the polytope, and by replacing x_i with y^{μ_i}, for each $i \in 1 \ldots d$. Equation (2.3) ensures that no factor in the denominator of Equation (2.2) becomes 0, and hence avoids introduction of singularities. Let

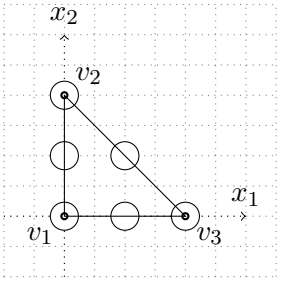
$$sub(r(P, x), y) \tag{2.4}$$

denote the result of the above substitution. Then, the constant term of the Laurent expansion of $sub(r(P, x), y)$ around $y = 1$ yields the desired count. Computing Laurent series expansions is a standard procedure and implemented, e.g., in Wolfram Alpha [92].

Example 1 *Consider the unimodular polytope*

$$P = (x_1 \geq 0 \wedge x_2 \geq 0 \wedge x_1 + x_2 \leq 2)$$

of dimension $d = 2$. *P has vertices* $v_1 = (0\ 0)$, $v_2 = (0\ 2)$, *and* $v_3 = (2\ 0)$ *and contains* 6 *integer points, as shown by the circles below.*



The generators of the vertex cones are given by

$$\begin{aligned} g_{v_1 1} &= (0\ 1) & g_{v_1 2} &= (1\ 0) \\ g_{v_2 1} &= (0\ -1) & g_{v_2 2} &= (1\ -1) \\ g_{v_3 1} &= (-1\ 0) & g_{v_3 2} &= (-1\ 1). \end{aligned}$$

Equation (2.2) *yields the following rational generating function* $r(P, x)$.

$$\frac{x_1^0 x_2^0}{(1-x_1^0 x_2^1)(1-x_1^1 x_2^0)} + \frac{x_1^0 x_2^2}{(1-x_1^0 x_2^{-1})(1-x_1^1 x_2^{-1})} + \frac{x_1^2 x_2^0}{(1-x_1^{-1} x_2^0)(1-x_1^{-1} x_2^1)}$$

Applying the substitution with $\mu = (-1\ 1)$ *yields the expression* $sub(r(P, x), y)$.

$$\frac{1}{(1-y)(1-y^{-1})} + \frac{y^2}{(1-y^{-1})(1-y^{-2})} + \frac{y^{-2}}{(1-y)(1-y^2)}$$

Computing the series expansion using the Wolfram Alpha command

$$series\ sub(r(P, x), y)\ at\ y = 1$$

produces

$$\cdots 5(y-1)^3 + 5(y-1)^2 + 6\ .$$

The constant coefficient 6 *yields the expected count.* ■

Symbolic cardinality expression The rational function representation of the generating function of a unimodular polytope shown in Equation 2.2 refers to the polytope's vertices and to the generators of its vertex cones. However, these generators and vertices do not have to be instantiated to concrete values in order for the evaluation of the generating function to be possible [91]. That is, the evaluation of the generating function can be carried out *symbolically* yielding a formula that expresses the cardinality of a polytope as a function of its generators, vertices, and a vector μ.

In our algorithm, we will use

$$\textsc{SymConeCard}(v, G, \mu) \tag{2.5}$$

to refer to the result of the symbolic evaluation of the generating function for the cone of a symbolic vertex v with generators G. By summing up $\textsc{SymConeCard}(v, G, \mu)$ for

all vertex cones we obtain a symbolic expression of the number of integer points in a symbolic polytope.[3]

Example 2 *The cardinality of a two-dimensional polytope with symbolic vertices* v_1, v_2, v_3 *and generators* $g_{v_i 1}$ *and* $g_{v_i 2}$*, with* $i \in 1..3$*, is given by*

$$\sum_{i=1}^{3} \textsc{SymConeCard}(v_i, \{g_{v_i 1}, g_{v_i 2}\}, \mu),$$

where

$$\begin{aligned}&\textsc{SymConeCard}(v_i, \{g_{v_i 1}, g_{v_i 2}\}, \mu)\\ &= (\mu_1^2 + 3\mu_1(\mu_2 - 2\mu_v - 1) + \mu_2^2 - 3\mu_2(2\mu_v + 1) + 6\mu_v^2 + 6\mu_v + 1)(12\mu_1\mu_2)^{-1}\\ &\quad \textit{with } \mu_1 = \mu \cdot g_{v_i 1},\ \mu_2 = \mu \cdot g_{v_i 2} \textit{ and } \mu_v = \mu \cdot v_i.\end{aligned}$$

■

Note that in order for $\textsc{SymConeCard}(v, G, \mu)$ to yield a valid count, the vertices and generators must satisfy a number of conditions, e.g., the symbolic cones need to be unimodular and the employed vector μ needs to satisfy Equation (2.3). We next present our interpolation procedure $\#\textsc{Itp}_{\textsc{LIA}}$ that creates constraints for ensuring these conditions.

2.4 Additional details on generating functions

In this section, we provide some additional (optional reading) details on generating functions.

More on unimodularity We give two alternative definitions of unimodularity.

Definition 1 *A cone is called unimodular if and only if its generators form a basis of* $\mathbb{Z}^d$*.*

Example 3 *The cone given by generators* $\left(\begin{smallmatrix}0\\1\end{smallmatrix}\right), \left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ *is unimodular. In contrast, the cone given by generators* $\left(\begin{smallmatrix}1\\2\end{smallmatrix}\right)$ *and* $\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ *is not unimodular since e.g.* $\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$ *cannot be represented as a positive linear combination of the generators.*

Equivalently, a cone is unimodular if and only if the parallelogram spanned by its generators contains only the origin. This parallelogram is called *parallelepiped*.

[3] This step relies on the fact that evaluating the generating function for each vertex cone separately and summing the results is equivalent to evaluating the sum of generating functions.

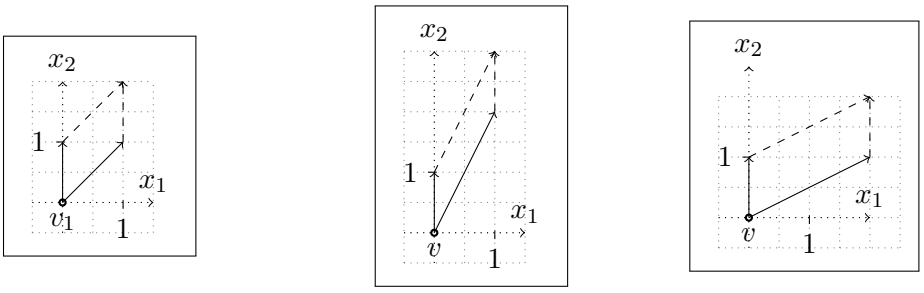


FIGURE 2.1: Parallelepipeds for two unimodular cones and one non-unimodular cone. The last parallelepiped contains integer point $(\,1\;1\,)$.

Definition 2 *The parallelepiped of a cone K with generators $g_1, \ldots, g_d$ is the set of points defined by*

$$\Pi_K = \{\textstyle\sum_{i=1}^{d} \alpha_i g_i \mid 0 \leq \alpha_i < 1\} \ . \tag{2.6}$$

Then cone K is unimodular if and only if Π_K contains exactly one integer point, namely, the origin. We provide examples in Figure 2.1.

More on rational generating functions We show that the generating function of a unimodular polytope can be represented by a short rational generating function. That is, we show that for a unimodular polytope P with vertices $\mathcal{V}$ and with generators $g_{v1}, \ldots, g_{vd}$, the following equality holds.

$$\sum_{m \in P \cap \mathbb{Z}^d} x^m = \sum_{v \in \mathcal{V}} \frac{x^v}{(1 - x^{g_{v1}}) \cdots (1 - x^{g_{vd}})} \ .$$

For this, we first give a formal statement of Brion's theorem.

Theorem 1 (Brion's Theorem [22]) The generating function of a polytope is equal to the sum of the generating functions of its vertex cones. Let $\mathcal{V}$ denote the set of vertices of polytope P, and let $cone(P, v)$ denote the vertex cone of P at vertex v. Then, the following holds.

$$f(P, x) = \textstyle\sum_{v \in \mathcal{V}} f(cone(P, v), x)$$

This result is perhaps surprising as each of the vertex cones contains an unbounded number of points. The intuition behind this equality is that the infinite parts of the vertex cones cancel each other out, leaving behind only the finite set of points in the polytope.

Using Brion's theorem, we just have to show that the generating function of each vertex cone can be efficiently represented. This follows from the next theorem.

Theorem 2 ([22]) For a unimodular cone K with generators $g_1, \ldots, g_d$, the following equality holds.

$$f(K, x) = \frac{x^v}{(1-x^{g_1})\cdots(1-x^{g_d})}$$

Proof 1 *Since the generators $g_1, \ldots, g_d$ form a basis of $\mathbb{Z}^d$ (see Definition 1), every integer point in K can be expressed as a positive linear combination of the generators.*

By the natural generalization of the geometric series

$$\sum_{(\alpha_1,\ldots,\alpha_d)\in\mathbb{Z}_+^d} x^{\alpha_1 g_1+\ldots+\alpha_d g_d} = \prod_{i=1}^{d} \frac{1}{(1-x^{g_i})}$$

the result follows.

2.5 Interpolation with cardinality constraints

In this section, we first define interpolation with cardinality constraints. Then we present the interpolation procedure $\#\textsc{Itp}_{\text{LIA}}$ that generates constraints on the cardinality of an interpolant and solves them using an SMT solver.

Cardinality interpolation

Let k be a variable and let w be a tuple of variables. Let φ and ψ be constraints in a given first-order theory. Then, a *cardinality constraint* is an expression of the form

$$\#\{w \mid \varphi\} = k \wedge \psi \tag{2.7}$$

where $\#\cdot$ denotes the set cardinality operator. We call the free variables of φ that do not occur in w *parameters*. A cardinality constraint is *parametric* if it has at least one parameter and *non-parametric* otherwise. The expression ψ is used to constrain the cardinality.

Example 4 *Consider the theory of linear integer arithmetic. The cardinality constraint*

$$\#\{x \mid 0 \leq x \leq 10\} = k \wedge k \leq 20$$

is non-parametric, whereas the constraint

$$\#\{x \mid 0 \leq x \leq n\} = k \wedge k \leq n + 1$$

is parametric in n. Both constraints are valid, since $\#\{x \mid 0 \leq x \leq 10\} = 11$ and $\#\{x \mid 0 \leq x \leq n\} = n + 1$. ■

Assume constraints φ^- and φ^+ such that φ^- implies φ^+. A *cardinality-constrained interpolant* for φ^-, φ^+, and cardinality constraint $\#\{w \mid \varphi\} = k \wedge \psi$ is a constraint φ such that

- 1) φ^- implies φ
- 2) φ implies φ^+, and
- 3) $\#\{w \mid \varphi\} = k \wedge \psi$ is valid.

For a parametric cardinality constraint, we say that the interpolation problem is parametric, and call it non-parametric otherwise.

Example 5 *Let $\varphi^- = (x = 0 \wedge n \geq 0)$ and $\varphi^+ = true$. Then*

$$\varphi = (0 \leq x \leq n)$$

is an interpolant that satisfies the cardinality constraint

$$\#\{x \mid \varphi\} = k \wedge k \leq n + 1 \ .$$

For $\varphi^- = false$, $\varphi^+ = x \geq 0$ and cardinality constraint $\#\{x \mid \varphi\} = k \wedge 1 \leq k \leq 10$, the constraint

$$\varphi = (0 \leq x \leq 5)$$

is a cardinality-constrained interpolant. ■

Note that our definition of interpolation differs from the standard, cardinality-free definition given e.g. in [72] in that we do not require the free variables in φ to be common to both φ^- and φ^+. We exclude this requirement because it appears to be overly restrictive for the setting with cardinalities, as the cardinality constraint imposes a lower/upper bound in addition to φ^- and φ^+. In particular, the common variables condition rules out both interpolants in Example 5, as the set of common variables is empty in both cases.

In this thesis, we focus on cardinality constraints with φ in linear arithmetic and ψ in (non-linear) arithmetic, which is an important combination for applications in software verification.

function $\#\textsc{Itp}_{\textsc{LIA}}(w, \varphi^-, \varphi^+, \psi, \textsc{Tmpl})$

1 $\textsc{Cons} := \mathit{true}$
2 $\textsc{SymCard} := 0$
3 $d :=$ length of w
4 $\mu :=$ vector of d fresh variables
5 $\mathcal{H}_\mathcal{V} := \bigcup\{\textsc{Tmpl}(v) \mid v \in \mathcal{V}\}$
6 **for each** $v \in \mathcal{V}$ **do**
7 $\quad\mathcal{H} := \textsc{Tmpl}(v)$
8 $\quad G := \emptyset$
9 $\quad$**for each** $H \in \mathcal{H}$ **do**
10 $\quad\quad g_{vH} :=$ vector of d fresh variables
11 $\quad\quad G := \{g_{vH}\} \cup G$
12 $\quad\textsc{Cons} := \textsc{Cons} \wedge \textsc{Vert}(v, \mathcal{H}, \mathcal{H}_\mathcal{V}) \wedge \textsc{Genr}(v, \mathcal{H}, G, \mu) \wedge \textsc{Unim}(v, G)$
13 $\quad\textsc{SymCard} := \textsc{SymCard} + \textsc{SymConeCard}(v, G, \mu)$
14 $\textsc{Cons} := \textsc{Cons} \wedge \textsc{Impl}(\varphi^-, \bigwedge \mathcal{H}_\mathcal{V}) \wedge \textsc{Impl}(\bigwedge \mathcal{H}_\mathcal{V}, \varphi^+)$
15 **return** $\textsc{SMTSolve}(\textsc{Cons} \wedge \textsc{Impl}(\textsc{SymCard} = k, \psi(k)))$

FIGURE 2.2: $\#\textsc{Itp}_{\textsc{LIA}}$ for cardinality constrained interpolantion for given $\textsc{Tmpl}$.

Interpolation algorithm

We present an algorithm $\#\textsc{Itp}_{\textsc{LIA}}$ for interpolation with cardinality constraints. For simplicity of exposition, we first consider the non-parametric case and discuss the parametric case in Section 2.6.

$\#\textsc{Itp}_{\textsc{LIA}}$ finds an interpolant φ in a space of polytope candidates that is defined through a template. This template is given by a function $\textsc{Tmpl}$ that maps a symbolic vertex $v \in \mathcal{V}$ to a set of symbolic hyperplanes that are determined to intersect in v, where each hyperplane $H \in \textsc{Tmpl}(v)$ is of the form

$$c_H \cdot w = \gamma_H \ .$$

The algorithm $\#\textsc{Itp}_{\textsc{LIA}}$ is described in Figure 2.2. It collects a constraint $\textsc{Cons}$ over the symbolic vertices and symbolic hyperplanes of φ, which ensures that any solution yields a unimodular polytope that satisfies conditions 1) – 3) of the definition of cardinality interpolation. In particular, $\#\textsc{Itp}_{\textsc{LIA}}$ ensures that the cardinality of φ satisfies ψ by constructing a symbolic expression $\textsc{SymCard}$ on the cardinality of φ in line 13, and requiring that this expression satisfies the cardinality constraint ψ in line 15. Line 12 produces well-formedness constraints $\textsc{Vert}(v, \mathcal{H}, \mathcal{H}_\mathcal{V})$ and $\textsc{Genr}(v, \mathcal{H}, G)$ that ensure a geometrically well-formed instantiation of the template $\textsc{Tmpl}$. The final conjunct in line 12 poses constraints on the generators of the vertex cones in φ that ensure their

unimodularity, as explained in Section 2.3. Finally, line 14 produces constraints that ensure the validity of the implications $\varphi^- \to \varphi$ and $\varphi \to \varphi^+$. The resulting constraint Cons is passed to an SMT solver that either returns a valuation of symbolic vertices and hyperplanes and hence determines φ, or fails.

Constraint generation We will now describe the constraint generation of #ItpLia in more detail. For each symbolic vertex v we make sure that it lies on the hyperplanes determined by Tmpl(v) and in the appropriate half-space with respect to the remaning hyperplanes. This is achieved by the following constraint.

$$\textsc{Vert}(v, \mathcal{H}, \mathcal{H}_\mathcal{V}) = \bigwedge_{H \in \mathcal{H}} c_H \cdot v = \gamma_H \ \wedge \bigwedge_{H \in \mathcal{H}_\mathcal{V} \setminus \mathcal{H}} c_H \cdot v < \gamma_H \tag{2.8}$$

By making the inequalities strict, we ensure that the polytope does not collapse into a single point, since in this case Brion's theorem does not hold.

SymConeCard and Unim refer to the generators of vertex cones determined by Tmpl. Hence we produce a constraint that defines these generators in terms of symbolic hyperplanes. Let g_{vH} denote the generator of the cone at vertex v that lies in the half-space described by hyperplane H. Then we constrain the generators of the cone at v as follows.

$$\begin{aligned}\textsc{Genr}(v, \mathcal{H}, G, \mu) = \textstyle\bigwedge_{H \in \mathcal{H}} (c_H \cdot g_{vH} \leq 0 \wedge \mu \cdot g_{vH} \neq 0 \\ \wedge \textstyle\bigwedge_{H' \in \mathcal{H} \setminus \{H\}} c_{H'} \cdot g_{vH} = 0)\end{aligned} \tag{2.9}$$

Here we require each generator g_{vH} to lie on the facet formed by the intersection of all hyperplanes $H' \in \mathcal{H} \setminus \{H\}$, and to point in the appropriate half-space wrt. H. Additionally the generator is constrained according to Equation 2.3. With the generators defined, we can ensure the unimodularity of vertex cones of the polytope by

$$\textsc{Unim}(v, G) = (abs(det(g_{vH_1}, \ldots, g_{vH_d})) = 1), \tag{2.10}$$

where $G = \{g_{vH_1}, \ldots, g_{vH_d}\}$. We then use SymConeCard(v, G, μ) to denote the counting expression of the symbolic cone of vertex v for our generators. We construct the counting expressions for the entire symbolic polytope φ by taking the sum over counting expressions for its vertex cones.

Finally, we generate constraints Impl for the implication conditions $\varphi^- \to \varphi$ and $\varphi \to \varphi^+$ by applying Farkas' lemma [84], which is a standard tool for such tasks [29, 80]. This

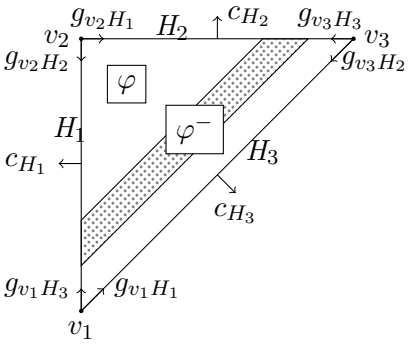


FIGURE 2.3: Illustration of Example 5.

lemma states that every linear consequence of a satisfiable set of linear inequalities can be obtained as a non-negative linear combination of these inequalities. Formally, if $Aw \leq b$ is satisfiable and $Aw \leq b$ implies $cw \leq \gamma$ then there exists $\lambda \geq 0$ such that $\lambda A = c$ and $\lambda b \leq \gamma$. When dealing with integers, Farkas' lemma is sound but not complete, see the discussion on completeness at the end of this section. Our implementation of IMPL handles non-conjunctive implication constraints by a standard method based on DNF conversion and Farkas' lemma.

Example 6 *Consider*

$$\varphi^- = (1 \leq x \wedge x - y \leq 1 \wedge\ x - y \geq -1 \wedge\ y \leq z \wedge z \leq 10),$$

$\varphi^+ = true$, $w = (x, y)$, *and* $\psi = (k \leq 120)$. *The solution* φ *is a polytope formed by three vertices* $\mathcal{V} = \{v_1, v_2, v_3\}$. *It is bounded by the supporting hyperplanes* $\mathcal{H}_\mathcal{V} = \{H_1, H_2, H_3\}$ *with normal vectors* c_{H_1}, c_{H_2} *and* c_{H_3}, *respectively. In our example, we use* TMPL *such that*

$$v_1 \mapsto \{H_1,\ H_3\},$$
$$v_2 \mapsto \{H_1, H_2\}\ \textit{and}$$
$$v_3 \mapsto \{H_2, H_3\}$$

restricting φ *to a triangular shape. Figure 2.3 shows* φ^-, *vertices, hyperplanes and a solution for* φ.

We obtain the following constraints:

$$\textsc{Vert}(v_1, \{H_1, H_3\}, \mathcal{H}_\mathcal{V}) = \begin{pmatrix} c_{H_1} \cdot v_1 = \gamma_{H_1} \wedge \\ c_{H_3} \cdot v_1 = \gamma_{H_3} \wedge \\ c_{H_2} \cdot v_1 < \gamma_{H_2} \end{pmatrix}$$

$$\textsc{Vert}(v_2, \{H_1, H_2\}, \mathcal{H}_\mathcal{V}) = \begin{pmatrix} c_{H_1} \cdot v_2 = \gamma_{H_1} \wedge \\ c_{H_2} \cdot v_2 = \gamma_{H_2} \wedge \\ c_{H_3} \cdot v_2 < \gamma_{H_3} \end{pmatrix}$$

$$\textsc{Vert}(v_3, \{H_2, H_3\}, \mathcal{H}_\mathcal{V}) = \begin{pmatrix} c_{H_2} \cdot v_3 = \gamma_{H_2} \wedge \\ c_{H_3} \cdot v_3 = \gamma_{H_3} \wedge \\ c_{H_1} \cdot v_3 < \gamma_{H_1} \end{pmatrix}$$

We get the following constraints on generators:

$$\textsc{Genr}(v_1, \{H_1, H_3\}, \{g_{v_1H_1}, g_{v_1H_3}\}, \mu) = \begin{pmatrix} c_{H_1} \cdot g_{v_1H_1} \leq 0 \wedge \\ c_{H_3} \cdot g_{v_1H_1} = 0 \wedge \\ c_{H_3} \cdot g_{v_1H_3} \leq 0 \wedge \\ c_{H_1} \cdot g_{v_1H_3} = 0 \end{pmatrix}$$

$$\textsc{Genr}(v_2, \{H_1, H_2\}, \{g_{v_2H_1}, g_{v_2H_2}\}, \mu) = \begin{pmatrix} c_{H_1} \cdot g_{v_2H_1} \leq 0 \wedge \\ c_{H_2} \cdot g_{v_2H_1} = 0 \wedge \\ c_{H_2} \cdot g_{v_2H_2} \leq 0 \wedge \\ c_{H_1} \cdot g_{v_2H_2} = 0 \end{pmatrix}$$

$$\textsc{Genr}(v_3, \{H_2, H_3\}, \{g_{v_3H_2}, g_{v_3H_3}\}, \mu) = \begin{pmatrix} c_{H_2} \cdot g_{v_3H_2} \leq 0 \wedge \\ c_{H_3} \cdot g_{v_3H_2} = 0 \wedge \\ c_{H_3} \cdot g_{v_3H_3} \leq 0 \wedge \\ c_{H_2} \cdot g_{v_3H_3} = 0 \end{pmatrix}$$

and unimodularity restrictions:

$$abs(det(g_{v_1H_1}, g_{v_1H_3})) = abs(det(g_{v_2H_1}, g_{v_2H_2})) = abs(det(g_{v_3H_2}, g_{v_3H_3})) = 1 \ .$$

The implication constraints in matrix notation are

$$\overbrace{\begin{pmatrix} -1 & 0 \\ 1 & -1 \\ -1 & 1 \\ 0 & 1 \end{pmatrix}}^{A} \begin{pmatrix} x \\ y \end{pmatrix} \leq \overbrace{\begin{pmatrix} -1 \\ 1 \\ 1 \\ 10 \end{pmatrix}}^{b} \quad \rightarrow \quad \overbrace{\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{pmatrix}}^{C} \begin{pmatrix} x \\ y \end{pmatrix} \leq \overbrace{\begin{pmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \end{pmatrix}}^{\gamma}$$

where, for each $i \in \{1, 2, 3\}$*, we obtain the following constraints for* H_i *by an application of Farkas' lemma:*

$$\exists \lambda^i : \lambda^i \geq 0 \wedge \lambda^i A = C_i \wedge \lambda^i b \leq \gamma_1 \ .$$

We pass the constraints to an SMT solver and obtain the solution

$$\varphi = (1 \leq x \ \wedge \ y \leq 10 \ \wedge \ y \geq x - 3)$$

with $\#\{(x, y) \mid \varphi\} = 91$. ■

Theorem 3 (Soundness) *If* $\#\textsc{Itp}_{\textsc{Lia}}(w, \varphi^-, \varphi^+, \psi, \textsc{Tmpl})$ *returns a solution* φ*, then* φ *is a cardinality-constrained interpolant for* φ^- *and* φ^+ *and cardinality constraint* $\#\{w \mid \varphi\} = k \wedge \psi$*.*

Proof 2 *We show that* φ *satisfies conditions 1) to 3). Conditions 1) and 2) follow from the use of Farkas' lemma. Since the conditions posed by* $\textsc{Vert}(v, \mathcal{H}, \mathcal{H}_\mathcal{V})$ *ensure that each vertex is active (part of the polytope) and that vertices are distinct, Brion's theorem is applicable and hence the generating function of* φ *can be expressed as the sum of the generating functions of its vertex cones. Each of* φ*'s vertex cones is unimodular by constraints* $\textsc{Unim}(v, G)$ *and its generating function is hence given by the expression in Equation 2.2. Summing over the evaluated rational generating functions of the vertex cones is equivalent to evaluating the sum of the rational generating functions by the fact that Laurent expansion distribute over sums. As a consequence, the expression* SymCard *corresponds to the cardinality of* φ *and, by the constraint in Line 15 in Figure 2.2, satisfies the cardinality constraint* ψ*.* □

Completeness For a given template, our method returns a solution whenever a solution expressed by the template exists, yet subject to the following two sources of incompleteness. First, solving non-linear integer arithmetic constraints is an undecidable problem and hence the call to SMTSolve may (soundly) fail. Second, Farkas'

lemma is incomplete over the integers. Note that these sources of incompleteness did not strike on benchmarks from the literature, see Section 2.8.

2.6 Interpolation with parametric cardinalities

We now briefly discuss the parametric interpolation problem by contrasting it with the non-parametric case. Computing the number of integer points in a polytope in terms of a parameter uses the techniques described in Section 2.3. The key challenge we face when extending cardinality-constrained interpolation to the parametric case is a quantifier alternation. While in the non-parametric case the constraints Cons are quantified as

$$\exists \mathcal{H}_\mathcal{V}\ \exists \mathcal{V} : \textsc{Cons},$$

introducing parameters changes the quantifier structure to

$$\exists \mathcal{H}_\mathcal{V}\ \forall p\ \exists \mathcal{V} : \textsc{Cons}$$

where p is a tuple of parameters in the cardinality constraint. The alternation stems from the fact that the parameter valuation detemines the intesection points, that is, the vertices, for parametric polytopes. This alternation has two implication on the computation of interpolants: first, for different values of p the number of vertices of a polytope can vary due to changes in the relative position of the bounding hyperplanes. As a consequence, templates with fixed number of vertices are only valid for a specific parameter range, which is called a *chamber* [91]. We deal with this aspect by considering a predicate *cmb* that restricts the parameter range to the appropriate chamber and that satisfies the implication constraints. We then conjoin *cmb* to the inferred polytope. [4]

Second, solving the cardinality constraint requires quantifier elimination for non-linear arithmetic. For this task we devise a constraint-based method ensuring positivity of a polynomial on a given range by referring to its roots.

Example 7 (Parametric counting) *Consider polytope*

$$Q = (x_1 \geq 0 \wedge x_2 \geq 0 \wedge x_1 + x_2 \leq n),$$

where the last equation is bounded by a parameter n rather than a constant. In this polytope, the coordinates of vertices v_2 and v_3 are linear expressions in the parameter n,

[4] Note that generators do not depend on the constant terms of thehyperplanes, which is why their constraints are not affected by variations in the parameters.

that is, for $n \geq 0$ we have $v_2 = (0\ n)$ and $v_3 = (n\ 0)$. Equation (2.2) yields the following generating function.

$$\frac{x_1^0 x_2^0}{(1-x_1^0 x_2^1)(1-x_1^1 x_2^0)} + \frac{x_1^0 x_2^n}{(1-x_1^0 x_2^{-1})(1-x_1^1 x_2^{-1})} + \frac{x_1^n x_2^0}{(1-x_1^{-1} x_2^0)(1-x_1^{-1} x_2^1)}$$

Applying the substitution and computing the series expansion yields the constant coefficient $(n^2+3n+2)/2$ which is an expression of number of integer points in Q in terms of the parameter n. ■

Example 8 (Parametric interpolation) *Consider again the interpolation problem from Section 2.2. We assume the following template where we fix some of the coefficients for simplicity of presentation (our algorithm deals with the general case):*

$$v_1 \mapsto \{H_1, H_4\},$$
$$v_2 \mapsto \{H_1, H_2\} \text{ and}$$
$$v_3 \mapsto \{H_2, H_3\}$$

with

$$H_1 = -i \leq 0$$
$$H_2 = a \cdot j \leq n + b$$
$$H_3 = i \leq 1$$
$$H_4 = i - j \leq 0\ .$$

We show exemplary vertex constraints for the parametric vertex $v_2 = (v_2^i\ v_2^j)$.

$$\forall n : \exists v_2^i, v_2^j : a \cdot v_2^j = n + b \wedge v_2^i = 1 \wedge v_2^i > 0 \wedge v_2^i < v_2^j$$

Note that these vertex constraints are valid only for n such that

$$2 \leq (n+b)/a,$$

which is when v_2 is active in the polytope. To ensure this we add a constraint

$$\forall n : cmb(n) \rightarrow 2 \leq (n+b)/a.$$

We add corresponding constraints for the other vertices of the template and further require that $cmb(n)$ be implied by the lower bound φ^-.

Evaluating the generating function (as described in Section 2.3) then yields the following expression on the cardinality of φ in terms of a and b

$$\begin{aligned}\textsc{SymCard}(\varphi) = \quad & (1/2 - 1/(2a^2)) \cdot n^2 + (-b/a^2 + b/a + 1/a + 1) \cdot n + \\ & (1 + 2b/a)\end{aligned} \tag{2.11}$$

The cardinality constraint on φ is given by

$$\exists a, b : \forall n : cmb(n) \rightarrow \textsc{SymCard}(\varphi) \leq \tfrac{(n+1)\cdot(n+2)}{2} \tag{2.12}$$

Solving the constraints yields $a = 1$, $b = 0$ and $cmb(n) = n \geq 2$. ■

Example 9 (Quantifier elimination) *Consider Equation 2.12 which provides an example constraint that we would like to solve. Our technique builds on the following observation: 2.12 is equivalent to*

$$\begin{aligned}\exists a, b : \forall n : \quad & cmb(n) \rightarrow \\ & 0 \leq ((1 - 2 \cdot c_2) \cdot n^2 + (3 - 2 \cdot c_1) \cdot n + (2 - 2 \cdot c_0))\end{aligned} \tag{2.13}$$

where c_2, c_1 and c_0 denote the coefficients of n in Equation 2.11. Let $p(n)$ denote the polynomial in Equation 2.13.

For simplicity of presentation, assume that $p(n)$ is of full degree and therefore has exactly two roots r_1 and r_2. Then these roots induce a partitioning of the domain of $p(n)$ such that $p(n)$ is either positive or negative throughout each partition. To ensure that Equation 2.13 holds, we then have to ensure that whenever $cmb(n)$ holds, $p(n)$ is positive.

Exploiting the following equality which is a consequence of the factor theorem which states that each polynomial $p(n)$ with root r contains a factor $(n - r)$

$$p(n) = (n - r_1) \cdot (n - r_2) \cdot k = k \cdot n^2 - k \cdot (r_1 + r_2) \cdot n + k \cdot r_1 \cdot r_2$$

we can now obtain a symbolic representation of the roots by equating the coefficients of the two polynomials. This yields:

$$1 - 2 \cdot c_2 = k \wedge 3 - 2 \cdot c_1 = k \cdot (-r1 - r2) \wedge 2 - 2 \cdot c_0 = k \cdot r_1 \cdot r_2.$$

Note that this step introduces a source of incompleteness as it restricts the solution space to polynomials with roots that can be expressed in the respective theory, i.e. integers or reals. Then we ensure that $p(n)$ is positive whenever $cmb(n)$ holds through the following

constraints

$$\begin{aligned} r1 \leq r2 \wedge ((cmb(n) \rightarrow n \leq r_1) \quad & \wedge 1 - 2 \cdot c_2 > 0) \quad \vee \\ ((cmb(n) \rightarrow r_1 \leq n \leq r_2) \quad & \wedge 1 - 2 \cdot c_2 < 0) \quad \vee \\ ((cmb(n) \rightarrow n \geq r_2) \quad & \wedge 1 - 2 \cdot c_2 > 0). \end{aligned}$$

Here, we ensure positivity on the respective partition by referring to the concavity of $p(n)$ *through its second derivative* $p''(n) = 1 - 2 \cdot c_2$.

Note that the above constraints are quantifier free. ■

2.7 Verification of programs with cardinality constraints

In this section, we sketch our algorithm #Horn for solving sets of Horn clauses with cardinality constraints. We choose Horn clauses as a basis for representing our verification conditions as they provide a uniform way to encode a variety of verification tasks [14, 15, 17, 43]. The interpolation procedure $\#\text{ITP}_{\text{LIA}}$ presented in Section 2.5 is a key ingredient for, but not restricted to, #Horn.

Horn clauses with cardinality constraints

A *Horn clause* is a formula of the form

$$\varphi_0 \wedge q_1 \wedge \cdots \wedge q_k \rightarrow H \tag{2.14}$$

where φ_0 is a linear arithmetic constraint, and $q_1, \ldots, q_k$ are uninterpreted predicates that we refer to as *queries*. We call the left-hand side of the implication *body* and the right-hand side *head* of the clause. H can either be a constraint φ, a query q, or a cardinality constraint of the form

$$\#\{w \mid q\} \leq \eta,$$

where η is a polynomial. By restricting cardinality constraints over queries to this shape, we ensure monotonicity, which is key for the soundness of over-approximation. For a clause $\varphi_0 \wedge q_1 \wedge \cdots \wedge q_k \rightarrow q$, we say that q *depends* on queries $q_1, \ldots, q_k$. We call a set of clauses *recursive* if the dependency relation contains a cycle, and *non-recursive* otherwise. For the semantics, we consider a *solution function* Σ that maps each query

symbol q occurring in a given set of clauses into a constraint. The satisfaction relation $\Sigma \models cl$ holds for a clause $cl = (\varphi_0 \wedge q_1 \wedge \cdots \wedge q_k \rightarrow H)$ iff the body of cl entails the head, after replacing each q by $\Sigma(q)$. The lifting from clauses to sets of clauses is canonical.

Algorithm description

#Horn takes as input a set C of recursive Horn clauses with cardinality constraints and produces as output either a solution to the clauses or a counterexample. Due the undecidability caused by recursion, #Horn may not terminate. The solver executes the following main steps: abstract inference, property checking, and refinement.

Abstract inference We iteratively build a solution for the set of *inference clauses*

$$\mathcal{I} = \{cl \in C \mid cl = (\ldots \rightarrow q)\}$$

by performing logical inference until a fixpoint is reached. This step uses abstraction to ensure that the inference terminates, where the abstraction is determined by a set of predicates *Preds*. This step is standard [43], as clauses $\mathcal{I}$ do not contain cardinality constraints.

Property checking We check whether the constructed solution satisfies all *property clauses* in

$$\mathcal{P} = C \setminus \mathcal{I} \ .$$

The novelty in #Horn is the check for satisfaction of cardinality constraints $\#\{w \mid \varphi\} \leq \eta$ where φ is a linear arithmetic constraint. Here we rely on a parametric extension of Barvinok's algorithm [91], which on input φ returns a set of tuples

$$\mathscr{B}(\varphi, w) = \{(cmb_1, c_1), \ \ldots\}$$

such that whenever the constraint cmb_i holds, the cardinality of $\#\{w \mid \varphi\}$ is given by the expression c_i, which may either be a polynomial c_i, or ω for the unbounded case. We hence reduce checking satisfaction of the cardinality constraint $\#\{w \mid \varphi\} \leq \eta$ to checking the following constraint.

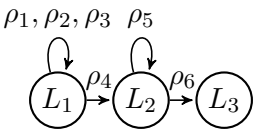
$$\textstyle\bigwedge_{(cmb,c)\in\mathscr{B}(\varphi,w)} (cmb \rightarrow c \leq \eta)$$

If the check succeeds, the algorithm returns the solution. Otherwise, the algorithm proceeds to a refinement phase to analyze the derivation that led to the violation of the property clause.

Refinement We construct a counterexample, i.e., a set *CEX* of recursion-free Horn clauses with cardinality constraints that represents the derivation that led to the violation of the property clause. This counterexample may either be genuine or spurious due to abstraction. To determine which it is, we rely on a solver for *non-recursive* clauses with cardinality constraints that either produces a solution for the clauses or reports that no such solution exists. If no solution exists, the algorithm returns the counterexample that represents a genuine error derivation. Otherwise it uses #ITP$_{\text{LIA}}$ to eliminate the cardinality constraint from the clauses thus producing a set of *cardinality-free* Horn clauses. We solve these clauses using existing methods [47] and obtain a set of predicates that we use to refine the abstraction.

Example 10 (Verification Conditions as Horn Clauses) *We consider a program that accesses a matrix stored in a dynamically allocated map m. The program manipulates the matrix through functions f and g. In the first loop, f is applied on a band around the diagonal, in the second loop, g is applied on the diagonal elements.*

```
    int c1=-1; int c2=-1;
L1: for(i=0; i<n; i++)
      for (j=0; j=<i; j++)
        if (i-j<3) {
          m(i, j) = f(i, j);
          c1=i; c2=j;
        }
L2: for(i=0; i<n; i++) {
      v = m(i, i);
      m(i, i) = g(v, i);
      c1=i; c2=i;
    }
L3:
```



Our goal is to prove a bound on the memory consumption. To make the reasoning more explicit, we instrument the program with auxiliary variables c_1 and c_2 that store the pairs of indices used to write into the map. Thus, by reasoning about the cardinality of the set of values (c_1, c_2) we track the memory consumption of the program.

Let the program variables be given by the vector $v = (i, j, c_1, c_2, n, pc)$ (we do not track m, f, g, v for space reasons) and the initial states of the program be described by the assertion

$$init(v) = (i = 0 \wedge j = 0 \wedge c_1 = -1 \wedge c_2 = -1 \wedge pc = L_1) \ .$$

In the control flow graph above, we collapse the control locations for the nested loop into a single program point L_1.

Some relevant transition relations are described below (we omit equalities over variables that stay unchanged, e.g., $pc' = pc$*).*

$$\rho_1(v, v') = (i < n \wedge j \leq i \wedge i - j < 3 \wedge j' = j + 1 \wedge c_1' = i \wedge c_2' = j)$$
$$\rho_2(v, v') = (i < n \wedge j \leq i \wedge i - j \geq 3 \wedge j' = j + 1)$$
$$\rho_3(v, v') = (i < n \wedge j > i \wedge j' = 0 \wedge i' = i + 1)$$

We represent the bound verification condition as the following set of recursive Horn clauses over query symbols $\mathcal{Q} = \{reach, index\}$*, where we let* $c = (c_1, c_2)$ *and* i *ranges between* 1 *and* 6.

$$\begin{array}{rll} cl_{init}: & init(v) & \rightarrow reach(v) \\ cl_i \ : & reach(v) \wedge \rho_i(v, v') & \rightarrow reach(v') \\ cl_{proj}: & reach(v) \wedge c_1 \geq 0 \wedge c_2 \geq 0 & \rightarrow index(c, n) \\ cl_{card}: & n \geq 0 & \rightarrow \#\{c \mid index(c, n))\} \leq 3n + 1 \end{array}$$

Query reach describes the set of reachable states and index describes the set of indices that were used for writing to the map. The clauses cl_{init}*, and* $cl_1, cl_2, \ldots$ *require the invariant reach to be inductive, i.e., that is is implied by initial states and preserved under the transition relation. The clause* cl_{proj} *projects reachable states on variables* c_1 *and* c_2*, and ensures that all reachable values of* c_1 *and* c_2 *(except for the negative initial values) are included in index. The clause* cl_{card} *encodes a cardinality constraint stating that the cardinality of the set of index values is bounded by* $3n+1$*. Finally, we note that the clauses are recursive, as e.g.* cl_1 *depends on itself.*

2.8 Experiments

We implemented our method in SICStus Prolog, and use its built-in constraint solver for the simplification and projection of linear constraints, HSF [43] for solving recursion- and cardinality-free Horn clauses, and Z3 [34] for non-linear/boolean constraint solving. We use BARVINOK [89] for checking whether a solution candidate satisfies a cardinality constraint. We use a 1.3 Ghz Intel Core i5 computer with 4 GB of RAM.

Benchmarks from the literature We use #HORN to analyze a set of examples taken from the recent literature on resource bound computation (in particular: time

Program	**Bound**	**Time**
Dis1 [46]	$max(n - x_0, 0) + max(m - y_0, 0)$	0.19s
Dis2 [46]	$n - x_0 + m - z_0$	0.17s
SimpleSingle [46]	n	0.11s
SequentialSingle [46]	n	0.11s
NestedSingle [46]	$n + 1$	0.15s
SimpleSingle2 [46]	$max(n, m)$	0.13s
SimpleMultiple [46]	$n + m$	0.16s
NestedMultiple [46]	$max(n - x_0, 0) + max(m - y_0, 0)$	0.08s
SimpleMultipleDep [46]	$n \cdot (m + 1)$	0.15s
NestedMultipleDep [46]	$n \cdot (m + 1)$	0.09s
IsortList [55]	$n^2 \cdot m$	0.19s
LCS [55]	$n \cdot x$	0.15s
Example 1 [95]	n	0.15s
Sum [60]	$2n + 6$	0.15s
Flatten [60]	$8l + 8$	0.13s

TABLE 2.1: Examples of resource bound verification [46, 55, 60, 95], with non-linear and disjunctive bounds on running time (the upper part of the tabe) and heap space usage (the lower part of the table), as well as imperative and functional programs. #HORN execution times are slightly faster than the literature. All bounds are precise.

and heap space), with results given in Table 2.1. We find that #HORN is able to prove all bounds in the literature while being slightly faster on average.

The time consumption of loops is bounded by synthesising a polytope containing all tuples of loop counter valuations. For example, for two loops with counters i and j bounded by parameters n and m, we synthesize a polytope of the form:

$$a \leq i \leq n + b \wedge c \leq j \leq m + d,$$

where a, b, c, d are inferred by our method. For heap consumption, we use the cost model of [60]. We encode *max* using disjunctions.

Dealing with relational dependencies We use #HORN to analyze programs `mcm` for matrix chain multiplication of Section 2.2 and *band matrix* from Example 10, with results in Table 2.2. These examples require the tracking of relational dependencies between variables. The example `mcm` is particularly challenging as it requires reasoning about recursive function calls. We are not aware of any other method that can handle programs with both features. We use a template specifying that the sought polytope consists of three and four symbolic vertices, respectively. Choosing a template that is not expressive enough might only allow to prove coarser bounds, however, one can automate the problem of finding an appropriate template by iterating over templates with an increasing number of symbolic vertices.

Program	Bound	Time
mcm	$\frac{(n+1)\cdot(n+2)}{2}$	0.6s
band matrix	$3n+1$	0.8s

TABLE 2.2: Examples tracking relational dependencies between variables.

Synthesis of countermeasures By relying on recursive Horn clauses as input language, #HORN is readily applicable to a number of verification questions that go beyond reachability. We illustrate this using the example of procedure `index`(a, e), which returns the first position of an element e in an array a.

```
int index(a, e) {
 int r=-1; t=0;
 for(i=0; r<0 && i<n; i++){
   if (a[i]==e) r=i;
   t++;
 }
 /* Padding */
 for(j=?; j<n, j++) t++;
 return r;
}
/* assert: bound cardinality of
   set of final values of t. */
```

Note that the execution time of `index` (modeled by the variable t) reveals the position of e. We apply #HORN for synthesizing a padding countermeasure against this timing side channel. Namely, we seek to instantiate the initialization of the variable j such that it provides enough padding for a given bound on leakage. This is achieved by bounding the cardinality of the set of possible final values of t. We add an additional clause that constrains the cardinality of values for t upon termination, as the logarithm of this number corresponds to the amount of leaked information in bits, see e.g. [85]. Table 2.3 provides the timings and synthesized initialization of j for different bounds on leakage.

Leakage bound, bits	Initialization	Time
$log(1)$	$j = i$	1s
$log(\frac{n}{2})$	$j = i + \frac{n}{2}$	0.7s
$log(\frac{n}{3})$	$j = \frac{2\cdot i+n}{3}$	0.7s

TABLE 2.3: Synthesis of countermeasures.

2.9 Related work

Counting integer points in polytopes The theory of counting integer points in polytopes has found wide-spread applications in program analysis. All applications we

are aware of (including [8, 41, 71, 91]) compute cardinalities for given polytopes, whereas our interpolation method computes polytopes for given cardinality constraints.

Verdoolaege et al. [91] also derive symbolic expressions for the number of integer points in parametric polytopes. In their approach, the parameter governs only the offset of the bounding hyperplanes (and hence the position of the vertices of the polytope) but not their tilt (and hence not the generators of the vertex cones). The advantage of fixing the vertex cones is that Barvinok's decomposition can be applied to handle arbitrary polytope shapes. In contrast, our interpolation procedure #ITP$_{\text{LIA}}$ (see Section 2.5) leaves the vertices *and* the generators of the vertex cones symbolic, up to constraints that ensure their unimodularity. The benefit of this approach is the additional degree of freedom for the synthesis procedure. #HORN leverages both approaches: the one from [91] for checking cardinality constraints, and #ITP$_{\text{LIA}}$ for refining the abstraction.

Recently, [41] presented a logic and decision procedure for satisfiability in the presence of cardinality constraints for the case of linear arithmetic. In contrast, we focus on synthesizing formulas that satisfy cardinality constraints, rather than checking their satisfiability.

Resource bounds In [68] a static analysis estimates the worst case execution time of non-parametric loops using the box domain. To ensure precision, the widening operator intersects the current abstraction with polytopes derived from conditional statements. In contrast, our approach generates abstraction consisting of parametric unimodular polytopes (which include boxes as a special case). In [46], the authors compute parametric resource and time bounds by instrumenting the program with (multiple-) counters, using static analysis to compute a bound for the counters, and combining the results to yield a bound for the entire program. In contrast, we fit polytopes over each iteration domain of the program, thus avoiding the need to infer counter placement and enabling higher precision by tracking dependencies between variables. In [90] a pattern-matching based method extracts polytopes representing the iteration domain of for-loops from C source. In contrast our method operates on unstructured programs represented as Horn clauses. In [60] and [55], a type system for the amortized analysis for higher-order, polymorphic programs is developed. Their focus lies on recursive data-types while we mostly deal with recursion/loops over the integers. In [24], this line of work is extended to the verification of C programs. In [3] and [75] closed-form bounds on resource usage are established by solving recurrence relations.

Quantitative verification Existing verification methods for other theories rely on cardinality extensions of SAT [42], or Boolean algebra of (uninterpreted) sets [65], multisets [77], and fractional collections [78]. These approaches focus on either computing the model size or checking satisfiability of a formula containing cardinality constraints. Cardinalities of uninterpreted sets are also used in [44] for establishing termination and memory usage bounds based on fixed abstractions. Finally, CEGAR approaches for weighted transition systems have been studied in [25] and [27]. These approaches considers abstractions for mean-payoff objectives such as limit-average or discounted sum.

2.10 Conclusion

We applied the theory of counting integer points in polytopes to devise an algorithm for a cardinality-constrained extension of Craig interpolation. This algorithm proceeds by posing constraints on a symbolic polytope that specify both its shape and cardinality and then solves the constraints via an SMT solver. We embedded our interpolation procedure into a solver for recursive Horn clauses with cardinality constraints and experimentally demonstrated its potential.

Chapter 3

Cardinality constraints for parametrized systems

3.1 Introduction

At the core of the majority of software systems we find concurrent and distributed components whose correct interplay is responsible for the proper functioning of the entire multitude of systems built on top. For example, (variations of) consensus protocols are powering cloud/data centers [23, 58], while on a single computer concurrent garbage collectors manage memory in modern runtime environments [73]. Verification of such core components is pursued by on-going industrial and academic efforts [48, 67, 76].

In many application domains concurrent/distributed systems can be formally modeled by so-called parameterized systems consisting of multiple copies of (almost) identical processes. Since properties of interest often need to hold for arbitrary number of participants, referring to the number of processes in a particular situation becomes an indispensable building block for proof construction. The following lemma from a classic textbook on multiprocessor programming [53] concisely illustrates this aspect while reasoning about a mutual exclusion protocol, which we discuss in more detail in Section 3.2:

> For j between 0 and $n-1$, there are at most $n-j$ threads at level (greater or equal to) j.

The lemma refers to the cardinality of the set of threads whose local state satisfies a certain condition, which could be formally written as

$$\#\{t \mid level(t) \geq j\} \leq n - j$$

where "#" represents the set cardinality operator. Cardinality is also ubiquitous in the realm of distributed protocols, as protocol descriptions routinely refer to the number of clients, replicas, faulty participants, or messages received from the leader [7].

Reasoning with cardinality for verifying parameterized systems is an under-researched topic, which started two decades ago within the deductive approach [70]. It required manual program annotation with invariants and offered their (semi-) automatic validity checking. Recently the topic was revisited: a programmer can now express invariants referring to cardinalities and quantifiers in a program logic [36] and check their validity using a corresponding semi-decision procedure. An emergent complementary line of research investigates how counting arguments can be integrated into a proof system that aims at automatic invariant discovery [39]. While these efforts are very promising, there is still a host of research challenges. Existing approaches either focus on sufficient expressiveness or automatic inference, but unfortunately not both at the same time. Perhaps the most important and intricate challenge is to keep track of the cardinality of sets defined by assertions in the logical theory of arrays combined with scalar data types, which is the lingua franca of parameterized systems verification [70].

In this chapter, we present a first step towards providing automatic inference for expressive cardinality-based verification of parameterized systems. We contribute #Π, an invariant synthesis method and its efficient implementation that can synthesize invariants tracking relations between 1) scalars, 2) cardinalities of sets represented using arrays, and 3) universally quantified array assertions. This previously unachieved combination facilitates fully automatic proofs of parameterized systems that were out of reach for automatic tools until now, as presented by examples in Section 3.7.

There are two key enablers for the effectiveness of our approach. First, we observe 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.

As the second enabler, we establish our synthesis procedure following the classic "constraint generation followed by solving" pipeline. It targets emerging Horn constraint solving technology as engine [43, 54, 56, 61, 64, 86, 87]. The key benefit we reap from relying on Horn solvers is the automation out-of-the-box for scalars. Besides that, we take advantage of existing algorithmic and engineering efforts and advances in solver development, which greatly contributes to the efficiency of #Π.

We implemented #Π and 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 invariants for intricate protocols. Several of them were verified automatically for the first time. For example, the filter lock protocol [53] requires infinitely many auxiliary counters tracking cardinality, which is handled by #Π via a cardinality constraint under universal quantification that keeps counters in an array. We also demonstrate that the ability of #Π to deal with cardinality does not incur any overhead when cardinality reasoning is not required, as #Π in general outperforms state-of-the-art tools on parameterized systems whose proofs are cardinality free.

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

3.2 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 3.1 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.

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

```
    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 3.1: Ticket lock.

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 3.2 shows a code fragment that implements the *filter lock*, a well-known mutual exclusion protocol [53]. We model this protocol using a cardinality constraint, in line 5. The protocol is based on the following idea:

```
    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 3.2: Filter lock.

- 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 $\#\{\texttt{t} \mid \texttt{lv(t)} > \texttt{i}\} = 0$ 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 $\#\{\texttt{t} \mid \texttt{lv(t)} = \texttt{i}\} \geq 2$ 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 3.3 shows code for the *one-third* rule [26, 36], 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 [26] 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 $HO(r)$.

```
   protocol oneThird
1:    instantiation x := v_0 with v_0 ≥ 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 3.3: One-third rule consensus protocol.

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.

3.3 Illustration

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.

$$init(a, pc) \quad \stackrel{\text{def}}{=} \quad (\forall t : pc(t) = 1) \wedge a = 0$$

$$next(a, pc, a', pc') \quad \stackrel{\text{def}}{=} \quad \exists t' : \begin{pmatrix} pc(t') = 1 \wedge \\ pc' = pc[t' \leftarrow 2] \wedge \\ a' = a + 1 \end{pmatrix}$$

$$safe(a, pc) \quad \stackrel{\text{def}}{=} \quad (\exists t : pc(t) > 1) \rightarrow a > 0$$

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.

$$\exists inv :$$

$$\begin{array}{lll} (a) & init(a, pc) & \rightarrow linv(a, pc) \\ (b) & inv(a, pc) \wedge next(a, pc, a', pc') & \rightarrow linv(a', pc') \\ (c) & inv(a, pc) & \rightarrow lsafe(a, pc) \end{array}$$

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.

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

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 [21], 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 yield 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.

$$(\#\{t \mid s(pc(t), a)\} = k) \wedge inv_0(pc, a, k)$$

$$init(lv, i, n) \stackrel{\text{def}}{=} (\forall t : lv(t) = 0 \wedge i(t) = 0) \wedge \#\{t \mid lv(t) = 0\} = n \wedge n \geq 2$$

$$next(lv, i, n, lv', i') \stackrel{\text{def}}{=} \exists t' : \left(\begin{array}{c} i(t') < n - 1 \wedge \\ (\#\{t \mid lv(t) > i(t')\} = 0 \vee \#\{t \mid lv(t) = i(t')\} \geq 2) \wedge \\ i' = i[t' \leftarrow (i(t') + 1)] \wedge lv' = lv[t' \leftarrow i'(t')] \end{array} \right)$$

$$safe(lv, i, n) \stackrel{\text{def}}{=} \#\{t \mid lv(t) = n - 1\} \leq 1$$

FIGURE 3.4: The filter-lock protocol as constraints.

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

$$\begin{aligned} &(\forall t : pc(t) = 1) \wedge \\ &\wedge ((\forall t : \neg s(pc(t), a)) \rightarrow k = 0) \wedge \ldots \\ &\rightarrow inv_0(pc, a, k) \end{aligned}$$

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

$$\begin{aligned} &pc(t) = 1 \\ &\wedge (\neg s(pc(t), a) \rightarrow k = 0) \wedge \ldots \\ &\rightarrow inv_0(pc, a, k) \ . \end{aligned}$$

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

$$\begin{aligned} s(pc(t), a) &\stackrel{\text{def}}{=} (pc(t) \geq 2) \\ inv_0(pc, a, k) &\stackrel{\text{def}}{=} (k \leq a) \ . \end{aligned}$$

3.4 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: $init(g, L)$, $next_{\text{T}}(g, l, g', l')$, 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 $next_{\mathrm{T}}(g, l, g', l')$ 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 $next_{\mathrm{T}}(g, l, g', l')$ 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.

$$next(g, L, g', L') \stackrel{\text{def}}{=} \exists t : \begin{pmatrix} next_{\mathrm{T}}(g, L(t), g', l') \wedge \\ L' = L[t \leftarrow l']) \end{pmatrix} \tag{3.1}$$

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 3.5 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 3.1 turns into a universal quantification.

The definitions above allow us to apply the standard proof rule for safety to describe a safe inductive invariant $inv(g, L)$ for the parameterized system. Since an instance of this proof rule is already shown in Section 3.3, 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.

Example 11 *Figure 3.4 shows initial states, transition relation and a safety property for the* filter-lock *protocol. Since each thread has a local variable i we represent i as an array that is indexed by a thread-id. We omit branches for the* `while` *and* `if` *statements that do not change the program state, and do not track the program counter for simplicity. Assertion init uses a cardinality constraint to encode the assumption that n threads participate in the protocol, and assertion next uses t' to denote the identifier of an arbitrary thread that tries to advance to a higher level.* ■

3.5 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 $\cdot[\cdot \leftarrow \cdot]$ for array updates (see e.g. [21] for more details). Let φ be a quantifier-free formula in that theory

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

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

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

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

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

(C) Rule CARD-UPD.

FIGURE 3.5: Rewriting rules for instantiating cardinality axioms.

such that φ 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, where we allow a restricted form of universal quantification over cardinality free formulas, such that a complete instantiation for the universal quantifiers can be efficiently computed (e.g. the array property fragment [21]).

Example 12 *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.

■

3.5.1 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 3.5 shows rewriting rules for instantiating cardinality axioms. Each rule specifies a rewriting 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 rules in more detail.

- Rule Card$_{\leq}$ instantiates a rule tracking *non-strict inequalities* between cardinalities.
- Rule Card$_{<}$ instantiates a rule tracking *strict 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 3.4, 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 t, the cardinality of a set referring to t is decremented if the t was part of the set before the update, and incremented if t is part of the set after the update. This is formalized through an indicator relation $\mathbb{1}$. For a set comprehension predicate $s(t, v)$, we define $\mathbb{1}$ as follows.

$$\mathbb{1}(\varphi, k) \stackrel{\text{def}}{=} (\varphi \wedge k = 1) \vee (\neg\varphi \wedge k = 0) \tag{3.2}$$

The rule Card-Upd can only be applied for formulas consisting of conjunctions, moreover defining formulas φ and φ' must be equivalent, except for the use of array f and g respectively. Finally, 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 12 (continued) *Consider again the formula*

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

Let

$$Def(k) = \#\{t \mid f(t) \geq 2\},$$

then, instantiating the axiom $\text{Card}_{\leq}$ *for a comparison with the empty set yields the following formula*

$$\begin{aligned}&(\forall t : f(t) = 1) \wedge \\ &((\forall t : f(t) \geq 2 \rightarrow \mathit{false}) \rightarrow k \leq 0) \wedge k \geq 1\end{aligned}$$

which we simplify (for readability) into

$$\begin{aligned}&(\forall t : f(t) = 1) \wedge \\ &(\exists t : f(t) \geq 2 \vee k \leq 0) \wedge k \geq 1\ .\end{aligned}$$

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

$$\begin{aligned}&f(t) = 1 \wedge \\ &(f(t) \geq 2 \vee k \leq 0) \wedge k \geq 1\ .\end{aligned}$$

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 4 (Soundness) *Axioms* $\text{Card}_{\leq}$, $\text{Card}_{<}$, *and* Card-Upd *are sound, i.e. the assertion under the line in Figure 3.5(a,b,c) is a logical consequence of* Δ.

Proof 3 *We prove soundness for the axiom* $\text{Card}_{\leq}$. *Consider a constraint*

$$\#\{t \mid \varphi(t)\} \leq \#\{t \mid \varphi'(t)\}\ .$$

This relationship holds if and only in there exists an injective homomorphism h *that maps each element in the left-hand set into an element of the right-hand set, i.e.*

$$\exists h : \begin{pmatrix} \forall t, t' : (h(t) = h(t') \rightarrow t = t') \wedge \\ \forall t : \varphi(t) \rightarrow \varphi'(h(t)) \end{pmatrix}$$

It is easy to see that this condition is implied by $\text{Card}_{\leq}$. □

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

- $\text{Card}_{\geq 0}$: 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. The following theorem states that instantiating the axiom CARD-UPD preserves difference bound constraints over cardinalities.

Theorem 5 (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.*

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

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

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

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 5, 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*

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

hold then

$$l = k + c \to \rho(k,l) \ .$$

Proof 4 (Theorem 5) *The "right-to-left" direction follows from the fact that*

$$\Psi \to \theta$$

holds. For the "left-to-right" direction assume that

$$\Psi \to \rho(k,l)$$

and θ hold, then we need to show $\rho(k,l)$. By case splitting over truth valuations for φ, and φ', we get

$$\theta \to l = k + c,$$

for some c. Then, from $\Psi \to \theta$, we can deduce that

$$\Psi \to l = k + c,$$

and by Proposition 1, we get that

$$l = k + c \to \rho(k,l)$$

from which $\rho(k,l)$ follows. □

Remark 1 *If for all cardinalities $\#\{t \mid \varphi\}$, we restrict occurrences of t in φ to 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.*

3.5.2 Venn decomposition

While for a number of our examples, the above axioms are sufficient (those in the upper table in Figure 3.8), for some examples (those in the lower table in Figure 3.8– in these examples comparison between cardinalities go beyond order constraints), we 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 $\mathit{region}(Q)$, which we define as follows.

$$\mathit{region}(Q) \stackrel{\text{def}}{=} \{t \mid \bigwedge_{p \in Q} p \wedge \bigwedge_{p \in (P \setminus Q)} \neg p\} \tag{3.3}$$

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

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

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

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

Example 13 *Consider the following constraint, which illustrates an argument in the verification of the* one-third *protocol presented in Section 3.2. This constraint is unsatisfiable, however the axioms of Section 3.5.1 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 following equations.*

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

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.* ■

3.6 The method #Π

We first give an overview of our method #Π which computes invariants for parametric systems by computing a solution (invariant) 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 3.6.1).
- *Quantifier elimination.* We then eliminate universal quantifiers that occur in the *invariant*. For this, we rely on existing methods [17, 57].
- *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 3.5.
- *Solving.* Finally, we employ an existing solver on the resulting clauses which yields the desired invariant.

3.6.1 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, the template defines an assertion $Shape(inv)$ of the following form, where 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.

$$\forall q_1, \ldots, q_n : \#\{t \mid s_1\} = k_1 \wedge \cdots \wedge \#\{t \mid s_m\} = k_m \wedge inv_0$$

Example 14 *In the filter-lock example, we search for an invariant with one quantifier and one cardinality expression defining an expression $Shape(inv)$ of the following form.*

$$Shape(inv) \quad \stackrel{\text{def}}{=} \quad \forall q : \#\{t \mid s\} \wedge inv_0$$

■

algorithm #Π
  input C, Q, *Shape* – clauses, queries, and shape template
  output Σ – solution function
  local
    function ElimCard – cardinality elimination (see Section 3.5)
    functions InstQ – quantifier instantiation
    function solve – Horn clause solver (see Section 3.6.2)
  begin
1: **foreach** $p \in dom(Shape)$ **and** $c \in C$ **do**
2: $c = c[Shape(p)/p]$
3: $c = \text{InstQ}(Shape(p), c)$
4: **end**
5: $C = \text{ElimCard}(C, Def)$
  end
  return solve(C, Q)
end

FIGURE 3.6: Algorithm #Π.

3.6.2 Solving

After performing quantifier instantiation for the universal quantifiers that occur in the shape template, #Π eliminates cardinalities from the clauses. The main difficulty in solving the resulting clauses stems from the fact that the unknown predicates that define set-comprehensions may occur negatively due to axiom instantiations. This aspect leads to non Horn constraints, which are out of scope for existing methods.

In order to deal with negative occurrences of unknown predicates, we provide *templates* for these predicates which specify their Boolean structure. For a predicate s over variables $x_1, \ldots, x_d$ a template consists of a formula

$$Tmp(s) \stackrel{\text{def}}{=} \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m} (?_1^{i,j} x_1 + ?_2^{i,j} x_2 + \cdots + ?_d^{i,j} x_d \leq ?^{i,j}) \tag{3.4}$$

where

$$?_1^{i,j}, \ldots, ?_d^{i,j}$$

are unknown parameters that are discovered by the solver. Fixing a template for an unknown predicate allows us to compute its negation which allows us to remove negative occurrences from the clauses. Note that template discovery can be automated by iteratively increasing the number of conjuncts/disjuncts.

To solve Horn clauses that contain unknown assertions under templates, we make use of an existing method (see [14, 15]). We will now briefly revisit this method. The method maintains a function INST, which assigns a formula (instance) satisfying the constraints specified in Tmp to each $p \in dom(Tmp)$. The instances are set to *true*, initially. The method starts by substituting each query by its instance and then invokes a standard Horn clause solver (e.g., [43, 54, 61]) on the resulting clauses. If the clauses are unsatisfiable and a counterexample is produced, the method picks a query (unknown assertion) under template, and invokes an SMT solver on the counterexample in order to find a new instance which avoids triggering the counterexample. If no such instance can be found for any query, the clauses are unsatisfiable, and the solver returns the counterexample. If a new instance was found for a query p, the constraints from the counterexample are added to $Tmp(p)$ in order to ensure that the same counterexample is not encountered in the future, and the process is repeated.

3.6.3 Algorithm #Π

Figure 3.6 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 as output an instantiated clause using existing instantiation methods [17, 57].

Function ELIMCARD(C, Def) takes as input a set of clauses and definition function Def and produces a set of cardinality-free clauses using the procedure described in Section 3.5.

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 and passes the resulting clauses to a Horn solver, which returns a solution function.

3.7 Evaluation

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

Cardinality-based reasoning

Table 3.8 summarises our results for reasoning with cardinalities. We use templates that specify the required number of quantifiers and set comprehension predicates, where the number of required conjuncts for the set comprehension predicates varies from 1 to 3. The upper table shows result on examples taken from [39]. 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. The example *cache* consists of a simple model of a cache-coherence protocol taken from [94], 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 3.8 contains the case studies from Section 3.2. We note that the ticket example 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 [39], 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.

Figure 3.7 contains code for the benchmark *garbage collection*, which consists of a simple model of a tri-colour mark-and-sweep garbage collector. 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

```
 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 3.7: Code for the benchmark *garbage collection*.

for elimination. We model this monotonicity property through 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 qualitative properties and cardinalities.

Program	**Card**	**Property**	**Inferred cardinalities**	**Time**
intro [39]	✓	$(\exists t : pc(t) = 2) \rightarrow b < a$	$\#\{t \mid pc(t) = 2\}$	1.2s
bluetooth [39]	✓	$(\exists t : pc(t) = 2) \rightarrow st = 0$	$\#\{t \mid pc(t) = 2\}$	1.6s
tree traverse [39]	×	$leaves = nodes + 1$	–	4.2s
cache [94]	✓	$\#\{t \mid pc(t) = 3\} \leq 1$	$\#\{t \mid pc(t) \geq 3\}$	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 [39]	$\#\{t \mid pc(t) = 3\} \leq 1$	$\#\{t \mid m(t) \leq s \wedge pc(t) = 2\}$, $\#\{t \mid pc(t) = 3\}$ $\#\{t \mid m(t) = q\}$	20.9s
filter lock [53]	$\#\{t \mid lv(t) = n - 1\} \leq 1$	$\#\{t \mid lv(t) \geq q\}$	27.5s
one-third rule [26, 36]	see Section 3.2	$\#\{t \mid x(t) = x(q)\}$	0.8s

FIGURE 3.8: Applying #Π to cardinality-based reasoning. Except ticket lock, all examples were automatically verified for the first time.

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 [1] and [83]. Table 3.9 summarises the results. Benchmarks in [1] 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's performance is on par with [1] when using a solver over the reals and slightly faster when solving over integers. Examples from [83] 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 [83] 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	**Quantifiers**	**Time**		
		#Π	Real [1]	Integer [1]
Simplified Bakery [1]	2	0.4s	0.8s	0.3s
Lamport's Bakery [1]	2	0.5s	2.1s	2s
Bogus Bakery [1]	2	0.6s	0.8s	11s
Ticket Mutex [1]	2	0.5s	0.3s	1.6s

Program	**Quantifiers**	**Time**			
		#Π	I[83]	P[83]	O[83]
barrier [83]	1	0.4s	0.1s	0.1s	0.1s
central barrier [83]	1	0.4s	0.1s	1.1s	6.2s
work stealing [40, 83]	1	0.5s	0.1s	0.1s	6.2s
dining philosophers [83]	0	8.2s	0.1s	6.3s	20s
robot 2x2 [83]	2	2.8s	0.2s	5.8s	1m45s
robot 2x3 [83]	2	16.1s	0.5s	16s	5m20s
robot 3x3 [83]	2	34.0s	0.9s	52s	19m28s
robot 4x4 [83]	2	TO	3.2s	5m3s	TO

FIGURE 3.9: Cardinality-free reasoning: results of comparing #Π to [1] and [83].

3.8 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 [65], generalized to multi-sets and fractional collections in [77, 78] and direct and inverse function/relation images in [93]. This logic is however not suitable for our purposes, as sets are uninterpreted. Hence the logic cannot be used for reasoning about sets 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.

Dragǒi et al. propose a logic that contains cardinality constraints over uninterpreted functions as well as limited quantifier alternation in [36]. This logic is geared towards the verification of consensus protocols such as Paxos [66] in the heard-of model [26] which allows for benign (communication) faults. While the logic is similar in spirit to our approach, [36] focuses on satisfiability checking in an expressive logic with the primary intent of *checking* inductive correctness arguments, whereas our focus lies on *synthesizing* such arguments automatically in a more restricted fragment.

An abstract interpretation based approach can track memory partition sizes [44] to infer memory usage properties. It relies on size tracking domain operations and can reason about data structures. An extension of such operations with the ability to track quantified array properties could lead to a viable alternative to our direct axiomatization.

Quantitative verification of parametric systems A classic example of the use of quantitative abstractions for parametric system is [79], where a number of bounded auxiliary counters for predefined sets of states are used to prove liveness of parametric protocols. The CIRC extension [50] of Blast [51, 52] shows how auxiliary counters can be inferred under predicate abstraction. [13] 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. These methods however do not support cardinalities with quantified array assertions.

Recently, Farzan et. al [39] 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, [39] is limited to the synthesis of scalar counters, whereas our method permits cardinalities that appear under a universal quantifier, which corresponds to synthesizing *arrays of counters* in their setting. Our experiments show that this ability is required for some of the more challenging benchmarks (i.e. the *filter* and the *ticket* example from Section 3.2).

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 this candidates can be done effectively [9, 37] even in the presence of complex communication topologies [10]. Our approach computes quantifier instantiation as a part of the inference process. In [62] 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 this thread is smaller than the variable m of all other threads" which enables verifying the ticket lock. In contrast, our method avoids tracking such dependencies by referring to the cardinality of the set of threads at a given location. [83] 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. Data flow graphs are used in [38] to separate reasoning about data and control and thus infer invariants that holds for arbitrary many threads. Our approach relies on transition relations, however, it would be interesting to see how the data flow graph perspective may be applied in our setting.

Universal quantification #Π directly benefits from the ability to synthesize quantified invariants to support cardinality reasoning. Dealing with universal quantification over arrays is a thriving research area that relies on abstract interpretation [74],

SMT [4, 5, 6], CLP [31, 32, 59], quantifier elimination and recurrence solving [35, 49], and first-order logic [63]. Our axiomatization of cardinality could potentially be used to put such methods to work on the discovery of cardinality based proofs.

Horn constraint solvers can be extended to support universal quantification by a form of local instantiation [17, 57]. Our approach can be seen as a direct extension with cardinality reasoning.

3.9 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 chapter presented #Π, a method and 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 #Π yielded an effective tool that is capable of verifying intricate parameterized systems using cardinality arguments, going beyond was possible with state-of-the-art methods. At the same time #Π is competitive or even outperforms 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 [45], 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 [81] appears to be a promising direction to consider.)

Chapter 4

Conclusion

This thesis introduced two methods for constructing invariants for programs that require reasoning about cardinalities.

Chapter 2 described #Horn, an invariant generation method that allows to prove bounds on the number of (subsets of) reachable states which has applications in resource bound analysis and quantitative information flow. Chapter 3 described #Π, an invariant generation method that allows to prove safety properties of parametrized systems, which has applications in multi-processor programming and distributed systems.

Our evaluation shows that both methods are practical and applicable to a wide range of verification tasks.

Bibliography

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

[2] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *CAV*, 2012.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *Programming Languages and Systems*, pages 157–172. Springer, 2007.

[4] 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), 2014.

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

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

[7] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. *ACM Trans. Comput. Syst.*, 2015.

[8] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *IEEE S& P*, 2009.

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

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

[11] A. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. In *FOCS*, 1993.

[12] A. Barvinok. *A Course in Convexity*. American Mathematical Society, 2002.

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

[14] T. A. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL*, 2014.

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

[16] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, 2011.

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

[18] N. Bjørner, K. L. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *SMT@IJCAR*, 2012.

[19] N. Bjørner, K. v. Gleissenthall, and A. Rybalchenko. Cardinalities and universal quantifiers for verifying parameterized systems. In *PLDI*, 2016.

[20] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, 2003.

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

[22] M. Brion. Points entiers dans les polyedres convexes. *Ann. Sci. Ecole Norm. Sup.*, 21(4), 1988.

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

[24] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional certified resource bounds. In *PLDI*, 2015.

[25] P. Cerny, T. A. Henzinger, and A. Radhakrishna. Quantitative abstraction refinement. In *POPL*. ACM, 2013.

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

[27] K. Chatterjee, A. Pavlogiannis, and Y. Velner. Quantitative interprocedural analysis. In *POPL*, 2015.

[28] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.

[29] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, 2003.

[30] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, second edition, 2001.

[31] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification using constraint handling rules and array constraint generalizations. In *VPT*, 2014.

[32] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying array programs by transforming verification conditions. In *VMCAI*, 2014.

[33] J. A. De Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. Effective lattice point counting in rational convex polytopes. *J. of Symb. Comp.*, 38(4), 2004.

[34] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[35] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, 2010.

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

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

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

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

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

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

[42] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. In *Handbook of Satisfiability*. 2009.

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

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

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

[46] S. Gulwani, K. K. Mehra, and T. Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *POPL*, 2009.

[47] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, 2011.

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

[49] T. A. Henzinger, T. Hottelier, L. Kovács, and A. Rybalchenko. Aligators for arrays (tool paper). In *LPAR*, 2010.

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

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

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

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

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

[55] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. In *POPL*, 2011.

[56] 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.

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

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

[59] J. Jaffar and A. E. Santosa. Recursive abstractions for parameterized systems. In *FM*, 2009.

[60] S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static determination of quantitative resource usage for higher-order programs. In *POPL*, 2010.

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

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

[63] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, 2009.

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

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

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

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

[68] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO*, 2009.

[69] L. Luu, S. Shinde, P. Saxena, and B. Demsky. A model counter for constraints over unbounded strings. In *PLDI*. ACM, 2014.

[70] Z. Manna and A. Pnueli. Verification of parameterized programs. In *Specification and Validation Methods*, pages 167–230. Oxford University Press, 1995.

[71] P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa. Dynamic enforcement of knowledge-based security policies. In *CSF*. IEEE, 2011.

[72] K. L. McMillan. An interpolating theorem prover. *TCS*, 2005.

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

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

[75] J. A. Navas, M. Méndez-Lojo, and M. V. Hermenegildo. User-definable resource usage bounds analysis for java bytecode. *Electr. Notes Theor. Comput. Sci.*, 253(5):65–82, 2009.

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

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

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

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

[80] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.

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

[82] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for Horn-clause verification. In *CAV*, 2013.

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

[84] A. Schrijver. *Theory of linear and integer programming.* Wiley, 1999.

[85] G. Smith. On the Foundations of Quantitative Information Flow. In *FoSSaCS*, 2009.

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

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

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

[89] S. Verdoolaege. Barvinok. http://freecode.com/projects/barvinok.

[90] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *IPACT*, 2012.

[91] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 2007.

[92] Wolfram. Wolfram alpha, series expansion. http://www.wolframalpha.com/examples/SeriesExpansions.html.

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

[94] 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*.

[95] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, 2011.