An SMT-based verification framework for software systems handling arrays

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Francesco Alberti

under the supervision of
Natasha Sharygina

April 2015

Dissertation Committee

Nikolaj Bjørner	Microsoft Research, Redmond, WA, USA
Rolf Krause	Università della Svizzera Italiana, Lugano, Switzerland
Viktor Kuncak	École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
Nate Nystrom	Università della Svizzera Italiana, Lugano, Switzerland

Dissertation accepted on 15 April 2015

Research Advisor	PhD Program Directors	
Natasha Sharygina	**Igor Pivkin**	**Stefan Wolf**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Francesco Alberti
Lugano, 15 April 2015

Abstract

Recent advances in the areas of automated reasoning and first-order theorem proving paved the way to the developing of effective tools for the rigorous formal analysis of computer systems. Nowadays many formal verification frameworks are built over highly engineered tools (SMT-solvers) implementing decision procedures for quantifier-free fragments of theories of interest for (dis)proving properties of software or hardware products.

The goal of this thesis is to go beyond the quantifier-free case and enable sound and effective solutions for the analysis of software systems requiring the usage of quantifiers. This is the case, for example, of software systems handling array variables, since meaningful properties about arrays (e.g., "the array is sorted") can be expressed only by exploiting quantification.

The first contribution of this thesis is the definition of a new Lazy Abstraction with Interpolants framework in which arrays can be handled in a natural manner. We identify a fragment of the theory of arrays admitting quantifier-free interpolation and provide an effective quantifier-free interpolation algorithm. The combination of this result with an important preprocessing technique allows the generation of the required quantified formulæ.

Second, we prove that accelerations, i.e., transitive closures, of an interesting class of relations over arrays are definable in the theory of arrays via $\exists^*\forall^*$-first order formulæ. We further show that the theoretical importance of this result has a practical relevance: Once the (problematic) nested quantifiers are suitably handled, acceleration offers a precise (not over-approximated) alternative to abstraction solutions.

Third, we present new decision procedures for quantified fragments of the theories of arrays. Our decision procedures are fully declarative, parametric in the theories describing the structure of the indexes and the elements of the arrays and orthogonal with respect to known results.

Fourth, by leveraging our new results on acceleration and decision procedures, we show that the problem of checking the safety of an important class of programs with arrays is fully decidable.

The thesis presents along with theoretical results practical engineering strategies for the effective implementation of a framework combining the aforementioned results: The declarative nature of our contributions allows for the definition of an integrated framework able to effectively check the safety of programs handling array variables while overcoming the individual limitations of the presented techniques.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer systems have a central role in modern society: almost all of today's industry depends critically on software either directly in the products or indirectly during the production, and the safety, cost-efficiency and environmentally friendliness of infrastructure, including the electric grid, public transportation, and health care, rely increasingly on correctly working hardware. The increasing role of computer systems in society means that the consequences of faults can be catastrophic. As a result proving the correctness of software is widely thought to be one of the most central challenges for computer science.

The aim of this thesis is developing theoretical frameworks, engineering techniques and computing infrastructures for the automatic, effective and rigorous analysis of software systems handling arrays. This is a highly challenging task, still out of reach for the modern state-of-the-art verification techniques.

1.1 Automated formal verification

Formal methods are nowadays gaining more and more importance in the area of software and hardware analysis from an academic and industrial perspective (see, e.g., [Lecomte, 2008, Newcombe, 2014]) given their ability to produce *proofs of correctness*. Formal methodologies for the analysis of software or hardware systems require a preliminary modeling phase where the system and its properties of interests are formally described. The analysis is subsequently performed on a logical-mathematical level. Formal methods are becoming integral part of the development process of computer systems, from requirement analysis to, of course, verification, as witnessed, for example, by the recent publication of the DO-333 document, *Formal Methods Supplement to DO-178C*

and DO-278A [RTCA, 2011], officially recognizing the use of formal methods as a means for certifying the dependability and reliance of computer systems in safety-critical domains.

We can distinguish two different groups of formal methodologies. One, generally called *deductive verification*, relies on tools like Coq, Isabelle/HOL, STeP, PVS, etc. This approach is interactive and is driven by the user's understanding of the system under validation. Fully automated *decision procedures* deal with some sub-problems (also called sub-goals) from which it is possible to infer the correctness of the system with respect to a given property. An advantage of deductive verification is that it can deal with infinite-state systems.

Another well-known formal method is *model-checking*. Model-checking was born as a technique for the verification of hardware systems [Clarke et al., 2001]. In this setting, the input system is formally represented by a transition system, modeling how the system reacts to external inputs and how it changes its internal configuration according to them. Properties of interests are formalized as temporal logic formulæ. Model-checking attracted, in the last three decades, the attention of academic and industrial worlds thanks to its distinguishing features. First, it is a completely automatic technique requiring no complex interactions with the user, after the preliminary stage of modeling the system. Second, whenever the analyzed system can exhibit a faulty execution, model-checking is able to produce the input values testifying such undesired behavior (called *counterexample*). Third, a model-checker performs an *exhaustive* exploration of all possible behaviors of the model: the reported absence of unwanted executions therefore ensures that no behaviors of the model can violate the given property.

The inventors of model checking E. M. Clarke, E. A. Emerson and J. Siphakis won in 2007 the prestigious *Turing Award*, witnessing the importance of this technology in the verification and validation of computer systems: "Their innovations transformed [model checking] approach from a theoretical technique to a highly effective verification technology that enables computer hardware and software engineers to find errors efficiently in complex system designs" [ACMs Press Release on the 2007 A.M. Turing Award recipients., 2007].

The weak-point of model-checking is that it can deal, in its original formulation, only with finite-state systems, i.e., systems admitting only finitely many reachable configurations. Even more, if the number of reachable configurations is huge, model-checking techniques might exhaust the available resources without being able to find whether the input system satisfies or not the given

property. This is generally called the *state-space explosion* problem. This problem has been tackled with the help of abstraction techniques. Abstraction involves, in general terms, loss of information. In our context, abstracting a system means removing all the details that are not relevant, or, better, are supposed not to be relevant, for generating a proof of correctness of the system. If details are removed, the reachable state-space of the system is more coarsely represented and its exploration needs, therefore, less resources. We can say that the "level of abstraction" of a system indicates the amount of details we are keeping. High level of abstraction indicates a coarse abstraction, where very few details of the system are kept, and a high reduction of the state-space. Low level of abstraction allows to know very precise (i.e., almost concrete) facts of the system, but might also result in a tiny reduction of the state-space with only little benefits for model-checking algorithms. Given well known undecidable results about program analysis, there cannot be an algorithm that outputs the correct level of abstraction to which abstracting a system for performing a proof of its correctness. This implies that the level of abstraction has to be either fixed a-priori, with the countereffect of reporting false alarms in case we select a too coarse abstraction, or iteratively refined until a proper one (if any) will be found. The price to pay in the latter case is admitting non-terminating analysis runs, always refining the level of abstraction. This thesis will fit within the latter schema.

Abstraction is one of the fundamental techniques adopted for the formal analysis of software systems. Software systems are generally checked against their assertions, i.e., the goal is to automatically infer whether a piece of code admits an execution violating one of its assertions[1]. These are all *safety inductive* properties. A proof of safety, in this case, is called *safe inductive invariant*. *Safe* means that the invariants describe (an over-approximation of) the sets of possible configurations of the analyzed program not including those which violates its assertions. *Inductive* means that the invariants include the starting configuration of the program and any computation starting from a configuration described by an invariant can only reach configurations still represented by such invariant.

The most widely adopted technique for abstracting a system is called *predicate abstraction* [Graf and Saïdi, 1997]. Predicates are quantifier-free formulæ over the set of variables handled by the system under verification. With predicate abstraction, the reachable configurations of the system are grouped together according to the predicates they satisfy. To make a parallel with the

[1]Code can have "implicit" assertions like division by zero, out-of-bound accesses, etc.

previous informal discussion about abstraction, the set of predicates induces the level of abstraction. Notably, if P is the set of predicates on which we are abstracting the system, the number of reachable states that have to be analyzed is reduced to at most $2^{|P|}$ states. Interestingly, predicate abstraction can be applied to systems admitting infinite reachable state-space. This is the case of software system, where infinitely many configurations are caused by the dynamic usage of memory.

If we are given a system $\mathcal{S}$ (representing a computer system) and a set of predicates P, an abstraction-based model-checking algorithm will start constructing an abstract system $\mathcal{S}^P$ in such a way that the set of possible executions of $\mathcal{S}$ is a sub-set of those of $\mathcal{S}^P$; the vice-versa does not hold. Thus, any safety property that holds for the executions of $\mathcal{S}^P$ also holds for those of $\mathcal{S}$. If there exists an *abstract counterexample*, i.e., an execution π^P of $\mathcal{S}^P$ not satisfying the property, we cannot directly conclude that there exists an execution of $\mathcal{S}$ violating the property. If π^P induces a *concrete counterexample* π of $\mathcal{S}$, then $\mathcal{S}$ has a bug, and π proves it. If such π does not exist, π^P is said to be a *spurious counterexample*, and $\mathcal{S}^P$ must be *refined*. This is the idea behind the CounterExample Based Abstraction Refinement framework, generally called CEGAR [Clarke et al., 2000]. Refinement works on the set of predicates P. The goal is to enlarge the set of predicates P to P' in such a way that the spurious counterexample π^P will become infeasible also in $\mathcal{S}^{P'}$.

Enabling effective and automatic refinement procedures is the main challenge of CEGAR. State-of-the-art refinement procedures exploit the spurious counterexample in order to find the new set P' excluding them from $\mathcal{S}^{P'}$. In particular, refinement is carried out with the help of Craig interpolants [Craig, 1957]. Given a spurious counterexample π^P, an interpolant-based refinement procedure generates a formula ϕ_{π^P} that is satisfiable iff π^P admits a feasible execution π of $\mathcal{S}$. If not, new predicates are computed as follows: Given a pair (A, B) of inconsistent formulæ, a Craig interpolant is a formula I built over the common vocabulary of A and B, entailed by A and unsatisfiable when put in conjunction with B. For refinement, the interpolant I may contain additional predicates and can be used to eliminate the part B of the counter-example that does not correspond to any execution of the concrete program while leaving A untouched. In this sense, in addition to discovering new predicates, the abstract program is refined *locally* by eliminating only part of the abstraction (namely, B) that gives rise to a counter-example. This is the idea behind the Lazy Abstraction with Interpolants framework [McMillan, 2006], which will play a central role in this thesis.

A complementary approach to abstraction is *acceleration*. This is another

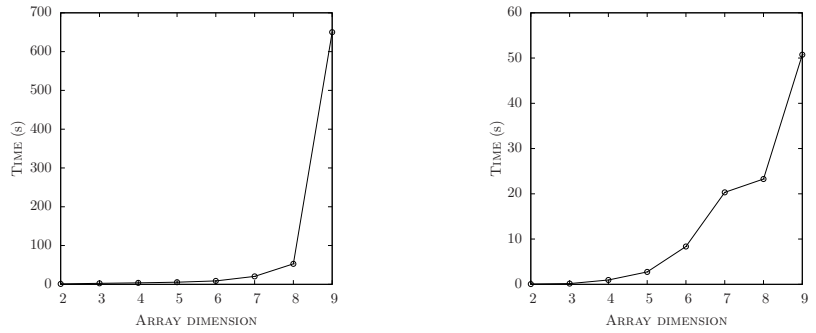
well-known technique in the model-checking literature. It relies on the generation of relations encoding the transitive closure of (part of) the transition relation symbolically encoding system evolution. Acceleration is applied to cyclic action of systems in order to find their reachable state-space in 'one shot'. This avoids divergence of the state-space exploration algorithm and provides a precise representation of the reachable state-space. The problem with acceleration is that transitive closure is a very powerful formalism that goes beyond first-order logic. This might prevent practical implementation of acceleration-based solutions for the analysis of real programs. On the other side, acceleration allows to prove important decidability results about software systems, as is has been done, for example, in [Bozga et al., 2014].

1.1.1 Challenges in automated formal verification for software handling arrays

The presence of array variables introduces a new level of complexity invalidating the effectiveness of the vast majority of the available software model-checking techniques. Let us consider, for example, a sorting procedure taken from [Armando et al., 2007b]:

```
void sort( int a[ ] , int N ) {
  int sw = 1;
  while ( sw ) {
    sw = 0;
    int i = 1;
    while  ( i < N ) {
      if ( a[i-1] > a[i] ) {
        int t = a[i];
        a[i] = a[i-1];
        a[i-1] = t;
        sw = 1;
      }
    }
    i = i + 1;
  }
}
```

We want to verify that, at the end of the procedure, the array `a` has been sorted. In order to do this, we need to add the piece of code



(a) (b)

Figure 1.1. Running times for CPAchecker and cbmc on the `sort` procedure.

```
for ( int x = 0 ; x < N ; x++ ) {
  for ( int y = x + 1 ; y < N ; y++ ) {
    assert( a[x] <= a[y] );
  }
}
```

Figure 1.1 reports running times for CPAchecker and cbmc[2] on the `sort` procedure for increasing values of the size N of the array a.

CPAchecker is a tool implementing (an advanced version of) the Lazy Abstraction with Interpolants approach. cbmc, instead, is one of the most efficient implementation of the *Bounded Model-Checking* (BMC) program analysis methodology [Biere et al., 1999]. This verification approach unrolls a bounded number κ of times the control-flow graph of a program searching for a feasible erroneous execution, i.e., an execution violating the program assertions. These unwinding of the control-flow graph are translated into a set of formulæ whose satisfiability implies that a bug is present in the code. These techniques are inherently incomplete, as they can only prove the presence or absence of bugs for executions with *bounded length* up to κ. Despite BMC is inherently incomplete, in some situations it is possible to establish a value κ which is sufficient to consider to guarantee the safety of executions of the program of arbitrary length. For the `sort` procedure, this number is $\kappa = \mathtt{N} + 1$.

[2]We selected cbmc and CPAchecker as they won the first and second place, respectively, of the overall category in the 3rd International Competition on Software Verification (SV-COMP'14, [Beyer, 2014]).

Unfortunately, there are no tools, to the best of our knowledge, dealing with array programs with the help of acceleration.

The graphics in Figure 1.1 show an exponential behavior for both tools[3]. BOOSTER, the tool implementing the results presented in this thesis that will be introduced in chapter 8, verifies the correctness of the `sort` procedure in 8.7 seconds, and does it for a version of the `sort` procedure where the parameter `N` is left parameterized. This means that the performance of BOOSTER does not depend on the actual value on `N` and does non change with different values for this symbolic constant.

The problem here is that the analysis of programs with arrays requires the ability to be able to reason in terms of quantified predicates. This is a non-trivial task invalidating many formal verification approaches. Indeed, the vast majority of state-of-the-art tools for the formal verification of infinite-state systems (e.g., [Cousot et al., 2005, Clarke et al., 2004, Beyer et al., 2007a, McMillan, 2006, Seghir et al., 2009, Hoder and Bjørner, 2012, Beyer and Keremoglu, 2011, Albarghouthi et al., 2012b]), beside implementing different analysis algorithms with innovative features, share the common limitation of working at a quantifier-free level. This is justified by the fact that logical engines like SMT-Solvers offer efficient decision procedures for quantifier-free fragments of theory of interests from a program analysis perspective. Only very recently some solvers started offering support for quantified fragments of theories with practical relevance (see, e.g., [Ge and de Moura, 2009, Ge et al., 2009, Reynolds et al., 2013, Bouton et al., 2009]). We expect that in the future always more tools for the analysis of computer systems will take advantage from this offer.

The problems with quantification is that decidable quantified fragments of mostly exploited theories might have a computational complexity prohibitive for interesting applications (e.g., quantifier elimination for Presburger Arithmetic is triple exponential [Oppen, 1978]) or might not admit a decision procedure at all. In fact, it is well known that (alternation of) quantifiers may easily lead to incompleteness results [Börger et al., 1997]. For these reasons, enabling automated formal techniques (dis)proving the safety of computer system at a level beyond the quantifier-free one is considered to be a major challenge.

In this thesis we will target the following open problems:

I. Identifying an abstraction framework suitable for generating and handling quantified predicates. In such a framework three requirements are

[3] CBMC (v4.3) has been executed with the option `--unwind` N`+1`, while for the CPACHECKER evaluation we enabled the options `-predicateAnalysis-PredAbsRefiner-ABEl-UF`; We would like to thank Dirk Beyer and his group for their support in running CPACHECKER.

mandatory:

- Identification of a fragment $\mathcal{F}$ of a first-order theory constituting a good trade-off between expressivity and efficiency. Such fragment has to be powerful enough in order to represent properties of interest for systems with arrays, and at the same time, being tractable from a computational point of view.
- Unsatisfiability of formulæ representing infeasible abstract counterexamples for systems handling arrays has to be decidable. This a key requirement for the lazy abstraction paradigm since it allows to detect if the abstract model has to be refined.
- The refinement procedure in charge to discover new predicates has to deal with quantified formulæ, i.e., has to synthesize quantified predicates. This is not a trivial task: quantifier-free refinement has the advantage of being highly incremental, meaning that the safe inductive invariant is build step after step from the refutation of single and concrete counterexamples. The "standard" theory of arrays does not admit quantifier-free interpolation [Kapur et al., 2006], and this problem kills such incrementality. Moreover, introduction of quantified predicates deserve a lot of attention since, in general, (alternation of) quantifiers may easily lead to undecidability results and therefore might prevent the availability of sound practical implementations. In our case, in addition, new predicates have to belong to the class $\mathcal{F}$ described above.

II. Investigating the definability of accelerations for relations over arrays and counters. A positive solution to this theoretical problem leads to another problem, i.e., how to apply acceleration to the verification of systems. In other words, in general, accelerations of relations encoding programs body – in our case those over arrays and counters – cannot be defined in first-order logic. Exceptional fragments of relations admitting firs-order definability of acceleration would likely allow it at some price (e.g., extra quantification), requiring further investigation in order to be practically exploited.

III. Identifying new decidable fragments of the theories of arrays. This investigation will be a natural follow-up to the achievements obtained as answers to the open problems of (II). Indeed, the positive theoretical result about the first-order definability of accelerations of relations of arrays and coun-

ters may lead to deeper results about the decidability of the safety of a class of programs with arrays.

IV. Analyzing interleaving strategies for a mutually beneficial integration of techniques for the generation of invariants addressing the points (I), (II) and (III) with other complementary static analysis solutions.

1.2 Contributions of the thesis

In this thesis we will work in a *declarative* setting, where the analysis of a computer system is performed by manipulating logical formulæ. This gives several advantages. First, the declarative formalism is close to real specifications and can be retrieved from them just by syntactic translations (as discussed in section 2.3). Moreover, our formal model is not specific for programs but encompasses more applications, e.g., from the parameterized verification of fault-tolerant protocols [Alberti et al., 2010a, Alberti et al., 2012d] to the analysis of access-control policies [Alberti et al., 2011a, Alberti et al., 2011b]. Second, the analysis frameworks we present in this thesis exploit existing technologies (SMT-solving) avoiding the need of ad-hoc implementations. In addition, they benefit both on the theoretical and practical level from advances in mathematical logic. Third, integrating several different and orthogonal analysis techniques (as we will do in chapter 8) is quite easy, and reduces only to the task of establishing the correct interfaces of the techniques collaborating in the analysis of the input programs.

We contribute to the area of formal verification of programs handling arrays with the following innovations:

- A quantifier-free interpolation procedure for a fragment of the theory of arrays, allowing for the extension of the Lazy Abstraction with Interpolants framework [McMillan, 2006] to a quantified level (chapter 3).
- A tool, SAFARI, implementing the above framework enhanced with heuristics to tune interpolation procedures and help convergence of the framework (chapter 4).
- The theoretical identification of a class of relations over arrays admitting definable first-order acceleration (modulo the theory of Presburger arithmetic enriched with free function symbols) (chapter 5, section 5.2).
- A pragmatic solution for exploiting acceleration procedures for the analysis of programs with arrays (chapter 5, section 5.3).

- The definition of a new decidable quantified fragment of the theories of arrays (chapter 6).
- The identification of class of programs with arrays admitting a decidable reachability analysis (chapter 7).
- A tool, BOOSTER, efficiently integrating all the aforementioned contributions in a single framework comprising, as well, other standard state-of-the-art static analysis procedures (chapter 8).

In the remaining part of the chapter we shall discuss in detail each of the contributions mentioned above.

1.2.1 Lazy Abstraction with Interpolants for Arrays

Lazy Abstraction with Interpolants (LAWI, [McMillan, 2006]) is one of the most efficient abstraction-based frameworks for the analysis of software systems. It is capable of tuning the abstraction by using different degrees of precision for different parts of the program by keeping track of both the control-flow graph, which describes *how* the program locations are traversed, and the data-flow, which describes *what* holds at a program location. The control-flow is represented explicitly, while the data-flow is symbolically encoded with quantifier-free first-order formulæ and it is subjected to abstraction. The procedure is therefore based on a *CounterExample Guided Abstraction Refinement* (CEGAR) loop [Clarke et al., 2000] in which the control-flow graph is iteratively unwound, and the data in the newly explored locations is overapproximated. When reaching an error location, if the path is spurious, i.e., the quantifier-free formula representing the manipulations of the data along the path is unsatisfiable, the abstraction along the path is refined. In state-of-the-art methods, this is done by means of interpolants [Henzinger et al., 2004, McMillan, 2006]. The procedure terminates when a non-spurious path is found, or when reaching an inductive invariant.

When arrays come into the picture, the situation is complicated by at least two problems. First, the need to handle quantified formulæ (as opposed to just quantifier-free) to take care of meaningful array properties; e.g., a typical post-condition of a sorting algorithm is the following universally quantified formula:

$$\forall i, j.\ (0 \leq i < j \leq a.length) \rightarrow a[i] \leq a[j]$$

expressing the fact that the array a is sorted, where $a.length$ represents the *symbolic* size of a. Second, the difficulty of computing quantifier-free inter-

polants. In [Kapur et al., 2006], it is shown that quantifiers must occur in interpolants of quantifier-free formulæ for the "standard" theory of arrays.

Research contribution. In chapter 3 of this thesis we describe a new verification approach that addresses the above problems. It redefines the lazy abstraction method based on interpolation, which is known to be one of the most effective approaches in program verification (section 3.2) and makes it possible to reason about arrays of unknown length by defining (i) an instantiating procedure to check the infeasibility of formulæ encoding counterexamples of array programs and (ii) a new quantifier-free interpolation procedure for a fragment of the theory of arrays suitable to represent counterexamples of array programs (section 3.3).

We also discuss, in chapter 4, the implementation strategies and the heuristics enabled in SAFARI, a tool implementing our new lazy abstraction with interpolants framework.

These results have been published in [Alberti et al., 2012a, Alberti et al., 2012c, Alberti et al., 2014a].

1.2.2 Acceleration techniques for relations over arrays

Acceleration is a well-known approach, orthogonal to abstraction, used to compute precisely the set of reachable states of a transition system. It requires to compute the transitive closure of relations expressing the system evolution. A major limitation reduces the applicability of acceleration. The problem is that, by definition, the transitive closure of a relation requires very powerful logical formalisms, like ones supporting infinite disjunctions or fixed points. Furthermore, for such expressive formalisms it is problematic to find efficient solvers (if any at all), which can be used in verification. To exploit acceleration in practice, therefore, it is required to identify special conditions on the relation making its transitive closure first-order definable within a suitable theory, like Presburger arithmetic. This is exactly the approach taken by relevant literature investigating numerical domains; when arrays come into the picture, additional complications arise, however, due to the fact that in order to model arrays one must enrich Presburger arithmetic with free function symbols. As a result, different – and more complicated – classes of transitions need to be handled.

Research contribution. Chapter 5 contributes the state-of-the-art of acceleration-based analysis techniques with both theoretical and practical new results.

First, we show that inside the theory of Presburger arithmetic augmented with free function symbols (added to model array variables), the acceleration of some classes of relations – corresponding, in our application domain, to relations involving arrays and counters – can be expressed in first order language. This result comes at a price of allowing nested quantifiers. Such nested quantification can be problematic in practical applications. To address this complication we show, as a second contribution, how to take care of the quantifiers added by the acceleration procedure: the idea is to import in this setting the so-called *monotonic abstraction* technique [Abdulla et al., 2007a, Abdulla et al., 2007b, Alberti et al., 2012d]. Third, we experimentally show that acceleration and abstraction have orthogonal strengths in the analysis of programs. This will lead to the establishment of the BOOSTER integrated framework described in chapter 8.

These results have been published in [Alberti et al., 2013b].

1.2.3 Decision procedures for Flat Array Properties

In the world of SMT-based static analysis solutions there is an increasing demand for procedures dealing with *quantified* fragments of the theories exploited in several applications, like $\mathcal{LIA}$ and the theories of arrays. Quantified formulæ are required in several tasks in verification, like modeling properties of the heap, asserting frame axioms, checking user defined assertions, defining axioms for extra theories and reason about parameterized systems.

The price for (universal[4]) quantification is often decidability: $\mathcal{EUF}$, the theory of Equality and Uninterpreted Functions, is only semi-decidable if we allow quantifiers. Satisfiability for universally quantified formulæ over $\mathcal{EUF} \cup \mathcal{LIA}$ is not even semi-decidable. Nonetheless, the problem of checking the satisfiability (modulo theories) of quantified formulæ got a lot of attention in the last years, with the goal of finding practical and efficient solutions to patterns of problems instead of focusing on more general – but likely less efficient – strategies. For some fragments of important theories it is possible to identify complete instantiation procedures [Bradley et al., 2006, Ge and de Moura, 2009]. Other theories of interest admit quantifier elimination procedures[5], as, for example, the theory of Linear Arithmetic over Integers [Cooper, 1972]. Remarkably, quantifier-elimination procedures can be exploited as decision procedures, but

[4] Usually a quantified formula is converted in NNF (Negative Normal Form) and then Skolemized. This returns an equisatisfiable formula with, at most, universal quantifiers.

[5] A theory T admits a quantifier elimination procedure iff for every formula φ it is possible to compute a T-equivalent quantifier-free formula φ'.

they rarely scale on big formulæ, since their complexity is usually high. For example, in the case of $\mathcal{LIA}$ a complexity bound for quantifier-elimination has been studied in [Oppen, 1978] and the best results in this area, to the best of our knowledge, are those recently reported by Bjørner in [Bjørner, 2010].

Outside such notable fragments, heuristics have to be designed to deal with quantifiers. The most implemented heuristic for handling universal quantifiers is the *matching modulo equalities* (E-matching) one [Detlefs et al., 2003]. This strategy exploit a given pattern to find suitable instances for the universally quantified variables. Instances of quantified formulæ are usually generated incrementally (since number of matches can be exponential). E-matching is not refutationally complete and, in practice, it requires "ground seed" to be applied[6]. E-matching, in general, can be adopted to handle quantifiers only if we are interested in checking the unsatisfiability of formulæ by providing good instantiation patterns increasing the chances to generate the required instances to detect the inconsistency.

Research contribution. In chapter 6 we will identify new decidable fragments of the monosorted and multisorted theories of arrays. We call the new decidable fragments *Flat Array Properties*, given that decidability is achieved by enforcing, among other restrictions, 'flatness' limitations on dereferencing, i.e., only positions named by variables are allowed in dereferencing.

We examine Flat Array Properties in two different settings. In one case, we consider Flat Array Properties over the theory of arrays generated by adding free function symbols to a given theory $\mathcal{T}$ modeling both indexes and elements of the arrays (section 6.2). In the other one, we take into account Flat Array Properties over a theory of arrays built by connecting two theories $\mathcal{T}_I$ and $\mathcal{T}_E$ describing the structure of indexes and elements (section 6.3). Our decidability results are fully declarative and parametric in the theories $\mathcal{T}, \mathcal{T}_I, \mathcal{T}_E$. For both settings, we provide sufficient conditions on $\mathcal{T}$ and $\mathcal{T}_I, \mathcal{T}_E$ for achieving the decidability of Flat Array Properties. Such hypotheses are widely met by theories of interest in practice, like Presburger arithmetic. Our decision procedures reduce the decidability of Flat Array Properties to the decidability of $\mathcal{T}$-formulæ in one case and $\mathcal{T}_I$- and $\mathcal{T}_E$-formulæ in the other case. We also analyze the complexity of our decision procedure when instantiated with theories of interests from a practical perspective (section 6.3.3).

These results have been published in [Alberti et al., 2014c, Alberti et al.,

[6]No way to check the unsatisfiability of $\forall x.p(x) \wedge \forall x.\neg p(x)$ by applying E-matching solutions.

2015].

1.2.4 Deciding the safety of a class of programs with arrays

Reachability analysis plays a crucial role in program verification. It relies on algorithms for computing the fixed point of the transition relation representing program's evolution and checking if any behavior leads to the violation of a given property. Since computation of the concrete fixed point is intractable in general, reachability analysis has been always coupled with solutions (like the abstraction-based ones listed above) devised to cope with the (unavoidable) divergence phenomena.

Research contribution. Chapter 7 is dedicated to the identification of a class of programs handling arrays or strings admitting decidable reachability analysis. Our result builds upon the results of chapters 5 and 6. The class of programs we identified includes non-recursive procedures implementing for instance searching, copying, comparing, initializing, replacing and testing functions.

These results have been published in [Alberti et al., 2014c, Alberti et al., 2013a, Alberti et al., 2015].

1.2.5 Booster: an acceleration-based verification framework for programs with arrays

Verifying the safety of software systems is a hard task. The generation of safe inductive invariants may require the cooperation of different techniques, each contributing with its own distinguishing features to the generation of the required proof of correctness for the input system. Combining different techniques for enabling the efficient analysis of complex inputs is a common practice in software verification (see, e.g., [Albarghouthi et al., 2012a, Henry et al., 2012]).

Research contribution. Starting from the experimental evaluation of section 5.4, we developed a tool, BOOSTER, combining the abovelisted contributions.

BOOSTER integrates abstraction frameworks like the one described in chapter 3 with standard abstraction-based invariant generation framework, e.g., *abstract interpretation* [Cousot and Cousot, 1977]. In addition, BOOSTER exploits acceleration in two different ways. Accelerations of loops falling in

decidable fragments are handled precisely, following the schema presented in chapter 6. Those requiring over-approximations and suitable refinement procedures (as discussed in chapter 5) are handled by an improved version of the MCMT model-checker [Alberti et al., 2014d], the fixpoint engine integrated in BOOSTER.

Moreover, BOOSTER integrates the abstraction and acceleration procedures in a verifying compiler framework nullifying the degree of user interaction and making the verification process completely automatic.

These results have been published in [Alberti et al., 2014d, Alberti et al., 2014b, Alberti and Monniaux, 2015].

Chapter 2

Background

This chapter introduces background concepts and terminology that is used in the rest of the thesis.

To make this document self-contained, we introduce in section 2.1 standard notions about syntax and semantics of first-order logic along with notational conventions we shall adopt in the thesis. Section 2.2 presents some theories of interest for the thesis and discusses some general undecidability results of the quantified fragment of the theory of arrays. Section 2.3 introduces the notion of *array-based transition system*, the formal model we will use to represent the computer systems of interest for this work.

2.1 Formal preliminaries and notational conventions

Definition 2.1.1 (Signature)**.** *A* signature Σ *is defined in terms of a set of sorts* $\{\sigma_1, \ldots, \sigma_n\}$*, a (possibly empty) set of function symbols and a (possibly empty) set of predicate symbols. Each function* f *and predicate* p *is endowed with an arity of the form* $\sigma_1 \times \cdots \times \sigma_n \to \sigma$ *and* $\sigma_1 \times \cdots \times \sigma_n$*, respectively.*

We assume that symbols are not overloaded. This means that each symbol is assigned to one and only one sort. We assume also the existance of the equality symbols $=_{\sigma_i}$, one for each sort of every signature Σ considered enceforth. Function symbols of arity 0 are called *(individual) constants*, predicates of arity 0 are *propositional constants*. Another assumption we make for every considered signature is the existence of sets of countably many variables V_{σ_i}, one for each sort.

Definition 2.1.2 (Term)**.** *A* Σ-term t *of sort* σ *is a variable in* V_σ*, a constant of sort* σ *or an expression of the kind* $f(t_1, \ldots, t_n)$*, where* f *is an* n*-ary function symbol from* Σ *of arity* $\sigma_1 \times \cdots \times \sigma_n \to \sigma$ *and each term* t_i *has sort* σ_i*.*

Definition 2.1.3 (Atoms)**.** *A* Σ-atom *(or* atomic formula*) is*

- *a propositional constant;*
- *an expression of the kind* $p(t_1, \ldots, t_n)$*, where* p *is a predicate of* Σ *of arity* $\sigma_1 \times \cdots \times \sigma_n$ *and each* t_i *is a* Σ*-term of sort* σ_i*;*
- *an expression of the kind* $t_1 =_\sigma t_2$*, where* t_1 *and* t_2 *are two terms of sort* σ*.*

Definition 2.1.4 (Formulae)**.** *A* Σ-formula *is a* Σ*-atom or an expression of the kind* $\neg\alpha$*,* $\alpha \wedge \beta$*,* $\forall_\sigma x.\alpha$*, where* α *and* β *are* Σ*-formulæ and* x *is a variable of sort* σ*.*

A *literal* is an atom or its negation. A *clause* is a disjunction of literals. A *ground* formula is a formula where variables do not occur. A formula without free variables is called a *sentence* (or a *closed formula*). A formula without quantifiers is called *quantifier-free*. We use lower-case Greek letters $\phi, \varphi, \psi, \ldots$ for quantifier-free formulæ and $\alpha, \beta, \ldots$ for arbitrary formulæ. We will use the standard Boolean abbreviations: "$\alpha \vee \beta$" stands for "$\neg\alpha \wedge \neg\beta$", "$\alpha \to \beta$" for "$\neg\alpha \vee \beta$", "$\alpha \leftrightarrow \beta$" for "$(\alpha \to \beta) \wedge (\beta \to \alpha)$", "$\exists_\sigma x.\alpha$" for "$\neg\forall_\sigma x.\neg\alpha$", where α and β are formulæ. $\top$ and $\bot$ represent the formulæ $\alpha \vee \neg\alpha$ and $\alpha \wedge \neg\alpha$, respectively, for any sentence α. When writing formulæ we will usually omit the brackets according to the following priorities: we stipulate that $\neg$ has the strongest priority and that $\wedge$ and $\vee$ have stronger priority than $\to$. An occurrence of a variable x in α is said to be *bounded* if it is within the scope of a quantifier of α, so if it is located in a sub-formula of α of the form $\forall_\sigma x.\beta$, otherwise x is said to be *free* in α. A variable is *free* in a formula α if it has at least one free occurrence in α. A *prenex formula* is a formula of the form $Q_1 x_1 \ldots Q_n x_n \varphi(x_1, \ldots, x_n)$, where $Q_i \in \{\exists, \forall\}$ and $x_1, \ldots, x_n$ are pairwise different variables. $Q_1 x_1 \cdots Q_n x_n$ is the *prefix* of the formula.

Notationally, variables will be denoted by lower-case Latin letters $x, a, i, e, \ldots$. Tuples of variables will be denoted by underlined letters $\underline{x}, \underline{a}, \underline{i}, \underline{e}, \ldots$ or bold face letters like $\mathbf{a}, \mathbf{v}, \ldots$. Bold face letters will be mainly used for tuples of variables which are generally fixed (e.g., variables handled by a program), underlined letters will denote tuples of variables which length may vary. For any variable v, v' is a primed copy of v. $v^{(n)}$ is a copy of v with n primes, $\mathbf{v}'$ is a

copy of $\mathbf{v}$ where every symbol has been primed and $\mathbf{v}^{(n)}$ is a copy of $\mathbf{v}$ where every symbol has n primes, for any tuple of variables $\mathbf{v}$. When we use $\mathbf{u} = \mathbf{v}$, we assume that two tuples have equal length, say n (i.e. $n := |\mathbf{u}| = |\mathbf{v}|$) and that $\mathbf{u} = \mathbf{v}$ abbreviates the formula $\bigwedge_{i=1}^{n} u_i = v_i$.

With $E(\underline{x})$ we denote that the syntactic expression (term, formula, tuple of terms or of formulæ) E contains at most the free variables in the tuple $\underline{x}$. Similarly, we may use $t(\mathbf{a}, \mathbf{s}, \underline{x}), \phi(\mathbf{a}, \mathbf{s}, \underline{x}), \ldots$ to mean that the term t or the quantifier-free formula ϕ have free variables included in $\underline{x}$ *and* that the free function and free constants symbols occurring in them are among $\mathbf{a}, \mathbf{s}$. Notations like $t(\mathbf{u}/\underline{x}), \phi(\mathbf{u}/\underline{x}), \ldots$ or $t(u_1/x_1, \ldots, u_n/x_n), \phi(u_1/x_1, \ldots, u_n/x_n), \ldots$ - or occasionally just $t(\mathbf{u}), \phi(\mathbf{u}), \ldots$ if confusion does not arise - are used for simultaneous substitutions within terms and formulæ.

Definition 2.1.5 (Structure)**.** *A* Σ-structure $\mathcal{M}$ *is a function having as a domain* Σ *and defined as follows:*

- *every sort symbol is associated to a non-empty set* $|\mathcal{M}|_{\sigma_i}$ *(the disjoint union of the* $|\mathcal{M}|_{\sigma_i}$*'s is called the* support *of* $\mathcal{M}$*);*
- *every predicate symbol* p *of arity* $\sigma_1 \times \cdots \times \sigma_n$ *is associated to a set* $p^{\mathcal{M}} \subseteq |\mathcal{M}|_{\sigma_1} \times \cdots \times |\mathcal{M}|_{\sigma_n}$*;*
- *every function symbol* f *of arity* $\sigma_1 \times \cdots \times \sigma_n \to \sigma$ *is associated to a* total *function* $f^{\mathcal{M}} : |\mathcal{M}|_{\sigma_1} \times \cdots \times |\mathcal{M}|_{\sigma_n} \to |\mathcal{M}|_{\sigma}$.

We denote with $\sigma^{\mathcal{M}}, f^{\mathcal{M}}, p^{\mathcal{M}}, \ldots$ the "interpretation" in $\mathcal{M}$ of the sort σ, the function symbol f and the predicate symbol p.

If Σ_0 is a sub-signature of Σ, the structure $\mathcal{M}_{|\Sigma_0}$ results from $\mathcal{M}$ by forgetting about the interpretation of the sort, the function and predicate symbols that are not in Σ_0 and $\mathcal{M}_{|\Sigma_0}$ is called the *reduct* of $\mathcal{M}$ to Σ_0.

Definition 2.1.6 (Assignment)**.** *Given a* Σ*-structure* $\mathcal{M}$*, a* Σ-assignment *is a function* $\mathtt{s}$ *mapping each* Σ*-term of sort* σ *to an element in the set* $|\mathcal{M}|_{\sigma}$*. It is defined as follows:*

$$\mathtt{s}(t) = \begin{cases} v \in |\mathcal{M}|_{\sigma} & \text{if } t \text{ is a variable of sort } \sigma \\ c^{\mathcal{M}} & \text{if } t \text{ is the constant } c \\ f^{\mathcal{M}}(\mathtt{s}(t_1), \ldots, \mathtt{s}(t_n)) & \text{if } t \text{ is of the kind } f(t_1, \ldots, t_n) \end{cases}$$

To avoid an unnecessary overloading of the notation, if confusion will not arise, we shall assume the well-sortedness of all the expressions and omit the specification of the sort symbols.

Definition 2.1.7 (Satisfiability)**.** *Given a signature Σ, a Σ-formula α and a Σ-structure $\mathcal{M}$, the relation $\mathcal{M}, \mathtt{s} \models \alpha$ ("α is true in $\mathcal{M}$ under the satisfying assignment $\mathtt{s}$") is inductively defined as follows:*

- $\mathcal{M}, \mathtt{s} \models t_1 = t_2$ *iff* $\mathtt{s}(t_1) = \mathtt{s}(t_2)$
- $\mathcal{M}, \mathtt{s} \models p(t_1, \ldots, t_n)$ *iff* $(\mathtt{s}(t_1), \ldots, \mathtt{s}(t_n)) \in p^{\mathcal{M}}$
- $\mathcal{M}, \mathtt{s} \models \neg\alpha$ *iff* $\mathcal{M}, \mathtt{s} \not\models \alpha$
- $\mathcal{M}, \mathtt{s} \models \alpha_1 \wedge \alpha_2$ *iff* $\mathcal{M}, \mathtt{s} \models \alpha_1$ *and* $\mathcal{M}, \mathtt{s} \models \alpha_2$
- $\mathcal{M}, \mathtt{s} \models \forall_\sigma x.\alpha$ *iff* $\mathcal{M}, \mathtt{s}_{\{x \mapsto v\}} \models \alpha$ *for every* $v \in |\mathcal{M}|_\sigma$

where $\mathtt{s}_{\{x \mapsto v\}}$ indicates that $\mathtt{s}$ maps the variable x to the symbol v.

Definition 2.1.8 (Theory)**.** *A* theory $\mathcal{T}$ *is a pair $(\Sigma, \mathcal{C})$, where Σ is a signature and $\mathcal{C}$ is a class of Σ-structures; the structures in $\mathcal{C}$ are called the* models *of $\mathcal{T}$.*

A Σ-formula ϕ is $\mathcal{T}$-satisfiable if there exists a Σ-structure $\mathcal{M}$ in $\mathcal{C}$ such that ϕ is true in $\mathcal{M}$ under a suitable assignment to the free variables of ϕ (in symbols, when φ is a sentence and no free variable assignment is needed, we write $\mathcal{M} \models \varphi$); it is $\mathcal{T}$-valid (in symbols, $\mathcal{T} \models \varphi$) if its negation is $\mathcal{T}$-unsatisfiable. Two formulæ φ_1 and φ_2 are $\mathcal{T}$-equisatisfiable iff if there exist a model of $\mathcal{T}$ and a free variable assignment in which φ_1 holds, then there exist a model of $\mathcal{T}$ and a free variable assignment in which also φ_2 holds, and vice-versa; they are $\mathcal{T}$-equivalent if $\varphi_1 \leftrightarrow \varphi_2$ is $\mathcal{T}$-valid; ψ_1 $\mathcal{T}$-entails ψ_2 (in symbols, $\psi_1 \models_{\mathcal{T}} \psi_2$) iff $\psi_1 \rightarrow \psi_2$ is $\mathcal{T}$-valid.

2.1.1 Quantifier-free interpolation and quantifier elimination

Two interesting properties of first-order theories are *quantifier-free interpolation* and *quantifier elimination*.

Definition 2.1.9 (Quantifier-free interpolation)**.** *A theory $\mathcal{T}$ has* quantifier-free interpolation *iff there exists an algorithm that, given two quantifier-free Σ-formulæ ϕ, ψ such that $\phi \wedge \psi$ is $\mathcal{T}$-unsatisfiable, returns a quantifier-free Σ-formula θ, the* interpolant*, such that: (i) $\phi \models_{\mathcal{T}} \theta$; (ii) $\theta \wedge \psi$ is $\mathcal{T}$-unsatisfiable; (iii) only the free variables common to ϕ and ψ occur in θ.*

Definition 2.1.10 (Quantifier elimination)**.** *A theory $\mathcal{T}$ admits* quantifier elimination *if and only if for any arbitrary Σ-formula $\alpha(\mathbf{x})$ it is always possible to compute a quantifier-free formula $\varphi(\mathbf{x})$ such that $\mathcal{T} \models \forall \mathbf{x}.(\alpha(\mathbf{x}) \leftrightarrow \varphi(\mathbf{x}))$.*

Quantifier-elimination implies quantifier-free interpolation. Let (A, B) be two inconsistent formulæ over a given signature. We can compute an interpolant for this pair by executing quantifier elimination to a formula I obtained by existentially quantifying the variables not belonging to B from the A formula.

In general, every pair of unsatisfiable formulæ admits an interpolant (see, e.g., [Hodges, 1993, §6.6]). Such interpolant is not ensured to be quantifier-free, though.

2.2 Satisfiability Modulo Theories

In the last two decades, many static analysis tasks strongly benefited from the advances in automated deduction and theorem proving. In particular, Satisfiability Modulo Theories (SMT) solving played – and still plays – a central role in several and heterogeneous solutions for program analysis and verification [de Moura and Bjørner, 2011].

Given a theory $\mathcal{T} = (\Sigma, \mathcal{C})$, the satisfiability modulo the theory $\mathcal{T}$ problem, in symbols $SMT(\mathcal{T})$, amounts to establishing the $\mathcal{T}$-satisfiability of *quantifier-free* Σ-formulæ. Given a theory $\mathcal{T}$ with decidable $SMT(\mathcal{T})$ problem, if $\mathcal{T}$ admits quantifier elimination, then $\mathcal{T}$ is decidable, i.e., the $\mathcal{T}$-satisfiability of *every* formula is decidable.

2.2.1 Examples of theories

In this thesis we will mainly work with programs handling arrays of integers. We will now introduce several theories that are particular relevant in this application domain, and discuss their properties (quantifier-elimination, interpolation, decidability of sub-fragments, etc.).

Enumerated data-type

The first theory we introduce is the mono-sorted theory of an *enumerated data-type* $\{e_1, ..., e_n\}$ in which the interpretation of the sort is a set of cardinality n, the signature of the theory contains only n constant symbols that are interpreted as the n distinct elements in the interpretation of the sort. The SMT problem for an enumerated data-type theory is decidable and every enumerated datatype theory has quantifier-free interpolation. As we will see, theories of enumerated-data types are useful to model the Boolean values (true and false) as well as the locations $l_0, ..., l_n$ of a program.

Linear Integer Arithmetic

A second relevant theory that will be used in this thesis is *Linear Integer Arithmetic*, $\mathcal{LIA}$. The signature of this theory has a single sort `INT`, the constants 0 and 1, the binary function symbols $+$ and $-$, the binary predicate $<$ and infinitely many unary predicates D_k, for each integer k greater than 1[1]. Semantically, the intended class of models for $\mathcal{LIA}$ contains just the structure whose support is the set of the integer numbers. `INT` is interpreted as $\mathbb{Z}$, the symbols $0, 1, +, -, <$ have the obvious interpretation over the integers and D_k is interpreted as the sets of integers divisible by k. The $SMT(\mathcal{LIA})$ problem is decidable and it is NP-complete [Papadimitriou, 1981]. In addition, $\mathcal{LIA}$ admits quantifier elimination (the extra predicates D_k are needed to get quantifier elimination [Oppen, 1978]).

Although $\mathcal{LIA}$ represents the fragment of arithmetic mostly used in formal approaches for the static analysis of systems, there are many other fragments that have quantifier elimination and can be quite useful; these fragments can be both weaker (like Integer Difference Logic, $\mathcal{IDL}$) and stronger (like the exponentiation extension of Semënov theorem) than $\mathcal{LIA}$.

Integer Difference Logic The theory $\mathcal{IDL}$ is a sub-theory of $\mathcal{LIA}$ whose atoms are written in the form $(0 \bowtie x - y + \bar{n})$, such that $\bowtie \in \{\leq, <, \neq, =, \geq, >\}$, and $\bar{n}$ is a numeral[2]. We can assume that the quantifier-free fragment of $\mathcal{IDL}$ is made by Boolean combinations of atoms of the kind $0 \leq y - x + \bar{n}$ [Sebastiani, 2007]. Decision procedures for the SMT($\mathcal{IDL}$) problem exploit the fact that a set of $\mathcal{IDL}$ quantifier-free atoms induces a graph with weighted edges of the kind $y \xrightarrow{\bar{n}} x$. The set of atoms is inconsistent iff the graph admits cycle of negative weight [Nieuwenhuis and Oliveras, 2005]. This shows that the satisfiability of a set of $\mathcal{IDL}$ quantifier-free atoms can be checked in polynomial time by adopting a standard algorithm for the analysis of graphs, e.g., the Floyd-Warshall algorithm [Floyd, 1962], having a complexity of $O(|V|^3)$, where V is the set of variables of the problem. $\mathcal{IDL}$ admits quantifier-free interpolation and quantifier-elimination (see, e.g., [Cimatti et al., 2010]).

Linear Integer Arithmetic with exponentiation Let $\exp_2$ be a unary function symbol that associates a number n to 2^n. The theory having as a signature

[1] Recall that we assumed the availability of the $=$ symbol for each sort of each signature we consider.

[2] The n^{th} numeral is the term $\underbrace{1 + 1 + \cdots + 1}_{n \text{ times}}$.

the set of symbols of $\mathcal{LIA}$ with the addition of $\exp_2$ is still decidable and admits quantifier-elimination [Semënov, 1984]. This implies that this theory admits quantifier-free interpolation.

Theory of arrays

Theories of arrays are theories parameterized in terms of the theories specifying the algebraic structures of the indexes and the elements of the arrays. There exist two ways of introducing arrays in a declarative setting, generating two different groups of theories: the *mono-sorted* theories of arrays, that will be denoted with $\mathtt{ARR}^1(\mathcal{T})$, and the *multi-sorted* theories of arrays, denoted with $\mathtt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$. The former is more expressive because (roughly speaking) it allows to consider indexes also as elements[3], but might be computationally more difficult to handle.

The following definition identifies a class of formulæ that will be exploited in the whole thesis.

Definition 2.2.1 (Flatness)**.** *Let $\mathcal{T}$ be a theory of arrays. An expression in the signature of $\mathcal{T}$ is said to be* flat *iff for every term of the kind $a(t)$ occurring in it (here a is a free function symbol), the sub-term t is always a variable.*

Notably, every formula admits an equisatisfiable flat counterpart, obtained by exploiting the rewriting rule

$$\phi(a(t), ...) \rightsquigarrow \exists x.(x = t \wedge \phi(a(x), ...))$$

Mono-sorted case

The mono-sorted theory $\mathtt{ARR}^1(\mathcal{T})$ *of arrays over* $\mathcal{T}$ is obtained from a given theory $\mathcal{T}$ by adding to it infinitely many (fresh) free unary function symbols. This means that the signature of $\mathtt{ARR}^1(\mathcal{T})$ is obtained from Σ by adding to it unary function symbols and that a structure $\mathcal{M}$ is a model of $\mathtt{ARR}^1(\mathcal{T})$ iff (once the interpretations of the extra function symbols are disregarded) it is a structure belonging to the original class $\mathcal{C}$.

Lemma 2.2.1. *Let $\mathcal{T}$ be a theory having decidable SMT problem. Then the SMT($\mathtt{ARR}^1(\mathcal{T})$) problem is decidable.*

[3]This is useful in the analysis of programs, where pointers to a heap region of the memory, modeled as an array m, are stored into a variable on the stack, i.e., are elements of m itself.

Proof. Let $\psi(\mathbf{x}, \mathbf{a}(\mathbf{x}))$ be a formula over the signature of $\mathtt{ARR}^1(\mathcal{T})$, where $|\mathbf{a}| = s$ and $|\mathbf{x}| = t$. We can assume that ψ is a flat formula. The Ackermann's expansion of ψ is obtained by replacing every function application with a fresh variable and by adding all the functional consistency constraints required. That is, consider a tuple $\mathbf{e} = \langle e_{k,l} \rangle$ of variables (of appropriate sort), with $1 \leq k \leq s$ and $1 \leq l \leq t$. The formula

$$\psi(\mathbf{x}, \mathbf{e}) \wedge \left[\bigwedge_{x_i, x_j \in \mathbf{x}} x_i = x_j \rightarrow \bigwedge_{k=1}^{|\mathbf{a}|} e_{k,i} = e_{k,j} \right]$$

is $\mathcal{T}$-satisfiable iff $\psi(\mathbf{x}, \mathbf{a}(\mathbf{x}))$ is $(\mathtt{ARR}^1(\mathcal{T}))$-satisfiable. □

An alternative to Ackermann's expansion for solving the SMT($\mathtt{ARR}^1(\mathcal{T})$) problem would be adopting a more general framework for checking the satisfiability of a quantifier-free formula over a theory obtained as a combination of two (or more) theories, e.g., [Nelson and Oppen, 1979]. As shown in [Bruttomesso et al., 2006], none of the two techniques is generally more efficient than the other.

Multi-sorted case

In order to build a multi-sorted theory of arrays, instead, we need two ingredient theories, $\mathcal{T}_I = (\Sigma_I, \mathcal{C}_I)$ and $\mathcal{T}_E = (\Sigma_E, \mathcal{C}_E)$. We can freely assume that Σ_I and Σ_E are disjoint (otherwise we can rename some symbols); for simplicity, we let both signatures be mono-sorted: let us call `INDEX` the unique sort of $\mathcal{T}_I$ and `ELEM` the unique sort of $\mathcal{T}_E$. The multi-sorted theory $\mathtt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$ *of arrays over* $\mathcal{T}_I$ *and* $\mathcal{T}_E$ is obtained from the union of $\Sigma_I \cup \Sigma_E$ by adding to it infinitely many (fresh) free unary function symbols (these new function symbols will have domain sort `INDEX` and codomain sort `ELEM`). The models of $\mathtt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$ are the structures whose reducts to the symbols of sorts `INDEX` and `ELEM` are models of $\mathcal{T}_I$ and $\mathcal{T}_E$, respectively.

Lemma 2.2.2. *Let $\mathcal{T}_I$ and $\mathcal{T}_E$ be two theories having decidable SMT problem. Then SMT($\mathtt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$) problem is decidable.*

Proof. Let ψ be a conjunction of literals in the signature of $\mathtt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$. We assume again that such literals are flat.

Let $\underline{i} = i_1, \ldots, i_n$ and $\underline{e}$ be the variables of sort `INDEX` and `ELEM`, respectively, occurring in ψ and let $\underline{a} = a_1, \ldots, a_s$ be the tuple of function symbols of ψ. By making case-splits, we can assume that ψ contain either $i = j$ or $i \neq j$ for

Property	Formula
Being initialized to a value v	$\forall x.(0 \leq x \wedge x < \mathsf{size}) \rightarrow a[x] = \mathsf{v}$
Not containing an element v	$\forall x.(0 \leq x \wedge x < \mathsf{size}) \rightarrow a[x] \neq \mathsf{v}$
Being equal to an array b (with $\|b\| = \|a\|$)	$\forall x.(0 \leq x \wedge x < \mathsf{size}) \rightarrow a[x] = b[x]$
Being sorted	$\forall x, y.(0 \leq x \wedge x < y \wedge y < \mathsf{size}) \rightarrow a[x] \leq a[y]$
Being reversed	$\forall x, y.(0 \leq x \wedge x < \mathsf{size} \wedge x + y = \mathsf{size}) \rightarrow a[x] = a[y]$

Table 2.1. Some properties of interest for an array a of length size.

all distinct $i, j \in \underline{i}$; in addition, in case $i = j$ is a conjunct of ψ, we can freely assume that $a_k(i) = a_k(j)$ is in ψ for all $a_k \in \underline{a}$.

We can further separate the literals whose root predicate symbol has argument of sort `INDEX` from the literals whose root predicate has arguments of sort `ELEM`, thus (from the way $\mathtt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$ is built) ψ can be rewritten as

$$\psi^I(\underline{i}) \wedge \psi^E(\underline{a}(\underline{i}), \underline{e}). \tag{2.1}$$

Let $\underline{d} = d_{11}, \ldots, d_{sn}$ be $s \times n$ fresh variables abstracting out the $\underline{a}(\underline{i})$: we claim that ψ is $\mathtt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$-satisfiable iff ψ^I is $\mathcal{T}_I$-satisfiable and ψ^E is $\mathcal{T}_E$-satisfiable. In fact, given models of ψ^I and ψ^E in the respective theories, it is easy to build a combined model for (2.1) by assigning to $a_k \in \underline{a}$ any function whose value on the element assigned to i_l is d_{kl} (the definition is correct because ψ contains a complete partition of the $\underline{i}$ and equalities have been propagated to ψ^E). □

The idea underlying the proof of Lemma 2.2.2 is to reduce the $\mathtt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$-satisfiability check of quantifier-free formulæ to $SMT(\mathcal{T}_I)$ and $SMT(\mathcal{T}_E)$ problems by using a unidirectional variant of the Nelson-Oppen combination schema [Nelson and Oppen, 1979]. The particularity is that only disjunctions of equalities between terms of sort `INDEX` are exchanged whereas those involving terms of sort `ELEM` are not.

2.2.2 General undecidability results for arrays of integers

In our context, we will mainly work with programs handling arrays of integers. We will, therefore, fix our background theory to be either $\mathtt{ARR}^1(\mathcal{LIA})$ or $\mathtt{ARR}^2(\mathcal{LIA}, \mathcal{LIA})$. The fragment of formulæ one would like to handle in this context is

$$\exists \mathbf{c}\, \forall \mathbf{i}\ \psi(\ \mathbf{c}\ ,\ \mathbf{i}\ ,\ \mathbf{a}(\mathbf{t}(\mathbf{i}))\ ) \tag{2.2}$$

because in this fragment one can express interesting properties for such programs, e.g., those reported in Table 2.1.

The fragment (2.2) does not admit, in general, a decision procedure. This constitute a limiting result that might prevent the practical development of tools able to solve the practical problem we target.

In the following paragraph we will prove this general negative result by showing that one can encode into a formula like (2.2) the reachability problem for 2-counters machines, also called *two registers Minsky machines*. Given that the reachability problem for such machines is undecidable, the fragment we are interested in cannot admit a decision procedure.

Minsky machines

Definition 2.2.2 (Two registers Minsky machine)**.** *A* two registers Minsky machine *is a finite set* $\mathbf{P}$ *of instructions for manipulating configurations seen as triples* (q, m, n)*, where* q *ranges over a finite set of locations* $L = \{l_1, \ldots, l_n\}$ *and* $m, n \in \mathbb{N}$*.*

The set I of instructions of a Minsky machines are the following:

- $q \to (r, 1, 0)$
- $q \to (r, 0, 1)$
- $q \to (r, -1, 0)[r']$
- $q \to (r, 0, -1)[r']$

These instructions modify the configuration of a Minsky machine. Let S be the set of triples $L \times \mathbb{N} \times \mathbb{N}$ and let

$$[\![\_]\!](\_) : I \times S \ \to \ S$$

be the function defining the semantics of the four instructions in I. For any $\tau \in I, q, q' \in L$ and $m, n, m', n' \in \mathbb{N}$, $[\![\tau]\!](\langle q, m, n\rangle) = \langle q', m', n'\rangle$ is defined as follows:

- if τ is "$q \to (r, 1, 0)$", then $q' = r, m' = m + 1, n' = n$;
- if τ is "$q \to (r, 0, 1)$", then $q' = r, m' = m, n' = n + 1$;
- if τ is "$q \to (r, -1, 0)[r']$" and $m \neq 0$, then $q' = r, m' = m - 1, n' = n$;
- if τ is "$q \to (r, -1, 0)[r']$" and $m = 0$, then $q' = r', m' = m, n' = n$;
- if τ is "$q \to (r, 0, -1)[r']$" and $n \neq 0$, then $q' = r, m' = m, n' = n - 1$;

- if τ is "$q \to (r, 0, -1)[r']$" and $n = 0$, then $q' = r', m' = m, n' = n$;

Definition 2.2.3 (n-steps reachability)**.** *Given a Minsky machine* $\mathbf{P}$ *and two configurations* $\langle q, m, n\rangle$ *and* $\langle q', m', n'\rangle$*,* $\langle q', m', n'\rangle$ *is* reachable by $\mathbf{P}$ from $\langle q', m', n'\rangle$ in n steps*, in symbols,*

$$\langle q, m, n\rangle \to^{n}_{\mathbf{P}} \langle q', m', n'\rangle$$

iff it is possible to reach from $\langle q, m, n\rangle$ *the configuration* $\langle q', m', n'\rangle$ *following the rules of* $\mathbf{P}$ *for at most* n *steps.*

Definition 2.2.4 (Reachability problem)**.** *Given a Minsky machine* $\mathbf{P}$ *and two configurations* $\langle q, m, n\rangle$ *and* $\langle q', m', n'\rangle$*,*

$$\langle q, m, n\rangle \to^{*}_{\mathbf{P}} \langle q', m', n'\rangle$$

is called (second) reachability (configuration) problem *and it holds iff it is possible to reach from* $\langle q, m, n\rangle$ *the configuration* $\langle q', m', n'\rangle$ *following the rules of* $\mathbf{P}$ *in any finite number of steps.*

Theorem 2.2.1 ([Minsky, 1967])**.** *The second reachability configuration problem for Minsky machines is undecidable.*

Undecidability result

Let $\mathcal{T}$ be the theory of Linear Integer Arithmetic $\mathcal{LIA}$ enriched with three free unary function symbols a_q, a_m and a_n. Every Minsky machine $\mathbf{P} = (\tau_1, \ldots, \tau_r)$ induces a formula $F_{\mathbf{P}}(i, a_q, a_m, a_n)$ of the kind

$$\tau_1(i, a_q, a_m, a_n) \vee \cdots \vee \tau_r(i, a_q, a_m, a_n)$$

where

- if τ_j is "$q \to (r, 1, 0)$", then

  $$\tau_j(i, a_q, a_m, a_n) \equiv a_q(i) = q \wedge a_q(i{+}1) = r \wedge a_m(i{+}1) = a_m(i){+}1 \wedge a_n(i{+}1) = a_n(i)$$

- if τ_j is "$q \to (r, 0, 1)$", then

  $$\tau_j(i, a_q, a_m, a_n) \equiv a_q(i) = q \wedge a_q(i{+}1) = r \wedge a_m(i{+}1) = a_m(i) \wedge a_n(i{+}1) = a_n(i){+}1$$

- if τ_j is "$q \to (r, -1, 0)[r']$" then

$$\tau_j(i, a_q, a_m, a_n) \equiv a_q(i) = q \wedge \left(\begin{array}{l} [a_m(i) \neq 0 \wedge a_q(i+1) = r \wedge a_m(i+1) = a_m(i) - 1 \wedge a_n(i+1) = a_n(i)] \vee \\ [a_m(i) = 0 \wedge a_q(i+1) = r' \wedge a_m(i+1) = a_m(i) \wedge a_n(i+1) = a_n(i)] \end{array} \right)$$

- if τ_j is "$q \to (r, 0, -1)[r']$" then

$$\tau_j(i, a_q, a_m, a_n) \equiv a_q(i) = q \wedge \left(\begin{array}{l} [a_m(i) \neq 0 \wedge a_q(i+1) = r \wedge a_m(i+1) = a_m(i) \wedge a_n(i+1) = a_n(i) - 1] \vee \\ [a_m(i) = 0 \wedge a_q(i+1) = r' \wedge a_m(i+1) = a_m(i) \wedge a_n(i+1) = a_n(i)] \end{array} \right)$$

Proposition 2.2.1. *Let* $\mathbf{P}$ *be a Minsky machine and* $\langle q_0, m_0, n_0 \rangle$, $\langle q_f, m_f, n_f \rangle$ *two configurations. The formula*

$$\begin{array}{l} a_q(0) = q_0 \wedge a_m(0) = m_0 \wedge a_n(0) = n_0 \wedge \\ \exists z. \left(\begin{array}{l} \forall i.((0 < i \wedge i < z) \to F_{\mathbf{P}}(i, a_q, a_m, a_n)) \wedge \\ a_q(z) = q_f \wedge a_m(z) = m_f \wedge a_n(z) = n_f \end{array} \right) \end{array} \tag{2.3}$$

is satisfiable iff $\langle q_0, m_0, n_0 \rangle \to^*_{\mathbf{P}} \langle q_f, m_f, n_f \rangle$.

Corollary 2.2.1. *The satisfiability of the fragment* (2.2) *of* $\mathtt{ARR}^1(\mathcal{LIA})$ *or* $\mathtt{ARR}^2(\mathcal{LIA}, \mathcal{LIA})$ *is undecidable.*

Notably, sub-fragments of (2.2) do admit a decision procedure. The decidable fragment described in [Bradley et al., 2006] does not include the (2.3) because of the $a(i+1)$ terms. (2.3) is also not included in the decidable fragment presented in [Habermehl et al., 2008b] because of the disjunctions in the $F_{\mathbf{P}}(i, a_q, a_m, a_n)$. In this thesis, we identify a third sub-fragment of (2.2) for both the mono-sorted and the multi-sorted theories of arrays not included with those presented in [Bradley et al., 2006, Habermehl et al., 2008b], called *Flat Array Properties*, that admits a decision procedure. Chapter 6 presents it and discusses in more details its comparison with the other known decidable sub-fragments of (2.2).

2.2.3 Definable function and predicate symbols

In the thesis we will use definable function and predicate symbols.

Definition 2.2.5 (Definable symbols)**.** *An n-ary predicate symbol P is defined in a theory $\mathcal{T}$ by a formula $\phi(\underline{x})$ not containing it iff we have $\mathcal{T} \models P(\underline{x}) \leftrightarrow \phi(\underline{x})$. Similarly, an n-ary function symbol f is defined in a theory $\mathcal{T}$ by a formula $\psi(\underline{x}, y)$ iff $\mathcal{T} \models f(\underline{x}) = y \leftrightarrow \psi(\underline{x}, y)$ and $\mathcal{T} \models \forall \underline{x}\, \exists! y.\phi(\underline{x}, y)$ ($\exists! y$ stands for 'there is a unique y such that ...').*

The addition of definable function and predicate symbols does not affect decidability of quantifier-free formulæ and can be used for various purposes, for instance in order to express directly array updates, case-defined functions, etc. For instance, if a is a unary free function symbol, the term $\mathsf{store}(a, i, x)$ (expressing the update of the array a at position i by over-writing x) is a definable function; formally, we have $\underline{x} := i, x, j$ and $\phi(\underline{x}, y)$ is given by $(j = i \wedge y = x) \vee (j \neq i \wedge y = a(j))$. This formula $\phi(\underline{j}, y)$ (and similar ones) is usually written as

$$y \;=\; (\mathsf{if}\ j = i\ \mathsf{then}\ x\ \mathsf{else}\ a(j))$$

to improve readability. Another useful definable function is integer division by a fixed natural number n: to show that integer division by n is definable, recall that in $\mathcal{LIA}$ the formula $\forall x\ \exists! y\ \bigvee_{r=0}^{n-1}(x = n * y + r)$ is valid.

2.3 Array-based transition systems and their safety

In this section we introduce the notion of *array-based transition system*, the formal model that we adopt to mathematically represent the computer system under verification.

2.3.1 Array-based transition systems

We start by introducing the concept of *guarded assignments in functional form*, class of relations that will be used to represent systems operations.

Definition 2.3.1 (Guarded assignments in functional form)**.** *Let $\mathcal{T}$ be a theory of arrays, $\mathbf{v} = \langle \mathbf{a}, \mathbf{s}, pc \rangle$ a tuple of symbols among which $\mathbf{a}$ is a tuple of free function symbols of $\mathcal{T}$ and $\langle \mathbf{s}, pc \rangle$ is a tuple of scalar variables, where pc is a variable which sort is interpreted as a finite set of values $\{l_1, \ldots, l_n\}$. A* guarded assignment in functional form *is a formula of the form*

$$\tau(\mathbf{v}, \mathbf{v}') := \exists \underline{k} \begin{pmatrix} pc = l_i \wedge pc' = l_j \;\wedge\; \phi_L(\underline{k}, \mathbf{a}, \mathbf{s}) \;\wedge \\ \mathbf{a}' = \lambda j.\, G(\underline{k}, j, \mathbf{a}, \mathbf{s}) \;\wedge\; \mathbf{s} = H(\underline{k}, \mathbf{a}, \mathbf{s}) \end{pmatrix} \tag{2.4}$$

where G and H are tuples of case-defined functions of length $|\mathbf{a}|$ and $|\mathbf{s}|$, respectively.

Notice that the use of λ-abstractions in (2.4) does not go beyond first-order logic, since $\mathbf{a}' = \lambda j.\ G(j, \dots)$ can be rewritten to the pure first-order formula $\forall j.\ \mathbf{a}'(j) = G(j, \dots)$. We will denote with the *matrix* of a guarded assignment in functional form the formula (2.4) itself without the existential prefix $\exists \underline{k}$; the *proper variables* of τ are those in $\underline{k}$.

Definition 2.3.2 (Array-based transition systems)**.** *Let $\mathcal{T}$ be a theory of arrays. An* array-based transition system (over $\mathcal{T}$) *is a tuple*

$$\mathcal{S}_{\mathcal{T}} = \langle \mathbf{v}; l_{\mathsf{init}}; l_{\mathsf{error}}; T \rangle \tag{2.5}$$

where, alike has been stated in the definition 2.3.1, $\mathbf{v} = \langle \mathbf{a}, \mathbf{s}, pc \rangle$: $\mathbf{a}, \mathbf{s}$ are the tuples of array variables and scalar variables, respectively, handled by the program and pc is variable taking values over a finite set $L = \{l_1, \dots, l_n\}$. l_{init} and l_{error} are two elements of L identifying the 'initial' and the 'error' location of the system. T is a finite set of guarded assignments in functional form $\{\tau_1(\mathbf{v}, \mathbf{v}'), \dots, \tau_r(\mathbf{v}, \mathbf{v}')\}$.

We assume the availability of two total functions $src : T \to L$ and $trg : T \to L$ identifying the 'source' and 'target' location for each $\tau \in T$. That is, for each $\tau \in T$, $\tau \models pc = src(\tau)$ and $\tau \models pc' = trg(\tau)$. In addition, by a little abuse of notation,

$$T(\mathbf{v}, \mathbf{v}') := \bigvee_{\tau \in T} \tau(\mathbf{v}, \mathbf{v}')$$

Finally, for all $\tau \in T, src(\tau) \neq l_{\mathsf{error}}$ and there exists at least one transition $\tau_i \in T$ such that $src(\tau_i) = l_{\mathsf{init}}$.

From programs to array-based transition systems

It is possible to associate an array-based transition system to the body of a procedure written in an imperative language by means of standard syntactical transformations. We illustrate the process on the procedure in Figure 2.1.

Let $\mathcal{T}$ be $\mathtt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$, where $\mathcal{T}_I$ is the mono-sorted theory $\mathcal{IDL}$ of integer difference logic (introduced in section 2.2.1) extended with a constant L. Let `INDEX` be the unique sort symbol of $\mathcal{T}_I$. The theory $\mathcal{T}_E$ is composed of three mono-sorted theories: one is $\mathcal{IDL}$ with a sort symbol called `ELEM`, another is the theory of the enumerated data-type of the Boolean values true and false, and the third one is the theory of the enumerated data-type of the locations

```
procedure Running( ) {
    i = 0;
    while ( i < L ) {
      if ( a[i] ≥ 0 ) b[i] = true;
      else b[i] = false;
      i = i + 1;
    }
    f = true; i = 0;
    while ( i < L ) {
      if ( a[i] ≥ 0 ∧ ¬b[i] ) f = false;
      if ( a[i] < 0 ∧ b[i] ) f = false;
      i = i + 1;
    }
    assert ( f );
}
```

Figure 2.1. The procedure Running.

l_0, l_1, l_2, l_3, l_4. Let `BOOL` and `LOC` be the sort symbols interpreted over the set $\{\mathsf{true}, \mathsf{false}\}$ and $\{l_0, l_1, l_2, l_3, l_4\}$, respectively. The tuple $\mathbf{a}$ of array state variables contains the function symbols a and b, interpreted as two functions from `INDEX` to `ELEM` and `INDEX` to `BOOL`, respectively. The tuple $\mathbf{c}$ of scalar variables contains the variables i of sort `INDEX`, pc of sort `LOC` and f of sort `BOOL`.

The following transitions $\tau_0, \ldots, \tau_9$ specify the instructions of the Running procedure. For the sake of readability, $mov(l_i, l_j)$ stands for $pc = l_i \wedge pc' = l_j$ and $id(t_1, \ldots, t_n)$ for $t_1 = t_1' \wedge \ldots \wedge t_n = t_n'$.

$\tau_0 := mov(l_0, l_1) \wedge i' = 0 \wedge id(a, b, f)$
$\tau_1 := mov(l_1, l_1) \wedge i < L \wedge a[i] \geq 0 \wedge i' = i + 1 \wedge b' = \mathsf{store}(b, i, \mathsf{true}) \wedge id(a, f)$
$\tau_2 := mov(l_1, l_1) \wedge i < L \wedge a[i] < 0 \wedge i' = i + 1 \wedge b' = \mathsf{store}(b, i, \mathsf{false}) \wedge id(a, f)$
$\tau_3 := mov(l_1, l_2) \wedge i \geq L \wedge i' = 0 \wedge f' = \mathsf{true} \wedge id(a, b)$
$\tau_4 := mov(l_2, l_2) \wedge i < L \wedge a[i] < 0 \wedge b[i] \wedge f' = \mathsf{false} \wedge i' = i + 1 \wedge id(a, b)$
$\tau_5 := mov(l_2, l_2) \wedge i < L \wedge a[i] \geq 0 \wedge \neg b[i] \wedge f' = \mathsf{false} \wedge i' = i + 1 \wedge id(a, b)$
$\tau_6 := mov(l_2, l_2) \wedge i < L \wedge a[i] \geq 0 \wedge b[i] \wedge i' = i + 1 \wedge id(a, b, f)$

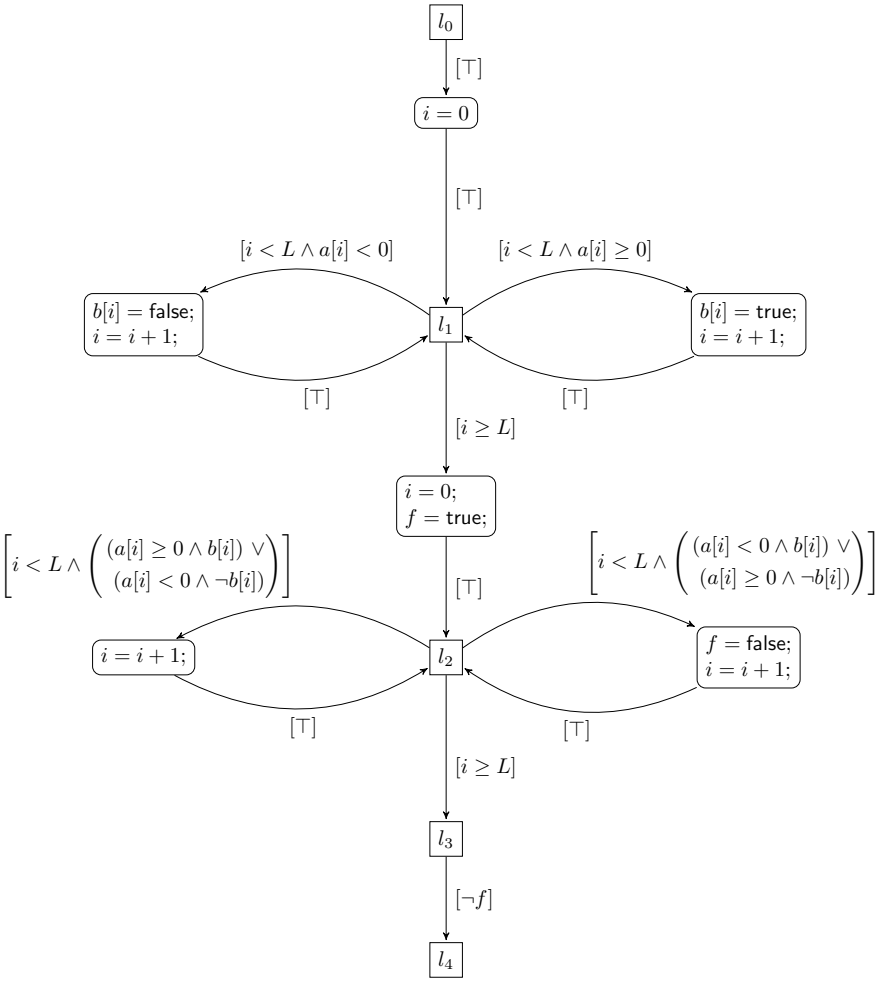


Figure 2.2. The control-flow graph of the procedure Running.

$$\tau_7 := mov(l_2, l_2) \wedge i < L \wedge a[i] < 0 \wedge \neg b[i] \wedge i' = i + 1 \wedge id(a, b, f)$$
$$\tau_8 := mov(l_2, l_3) \wedge i \geq L \wedge id(a, b, i, f)$$
$$\tau_9 := mov(l_3, l_4) \wedge \neg f \wedge id(a, b, i, f)$$

where, as stated in section 2.2.3, $\mathsf{store}(b, i, e)$ abbreviates the expression $\lambda j.\mathsf{if}\ (j = i)\ \mathsf{then}\ e\ \mathsf{else}\ b[j]$.

We are left to specify the initial l_{init} and error l_{error} locations. For the procedure Running in Figure 2.1, we define $l_{\mathsf{init}} = l_0$ and $l_{\mathsf{error}} = l_4$.

2.4 Safety and invariants

In this thesis we consider only safety invariant properties: given an array-based transition system $\mathcal{S}_\mathcal{T}$, we are interested in checking whether its error location is reachable. The notion of *safe transition system* is formalized as follows.

Definition 2.4.1 (Safety)**.** *A transition system* $\mathcal{S}_\mathcal{T} = \langle \mathbf{v}, l_{\mathsf{init}}, l_{\mathsf{error}}, T \rangle$ *is* safe *iff the following formula*

$$pc^{(n)} = l_{\mathsf{init}} \ \wedge \ \bigwedge_{i=1}^{n} T(\mathbf{v}^{(i)}, \mathbf{v}^{(i-1)}) \ \wedge \ pc^{(0)} = l_{\mathsf{error}} \tag{2.6}$$

is $\mathcal{T}$-unsatisfiable for every $n \geq 0$.

In our framework, the verification of a safety property for an imperative program P can be reduced to check the reachability of the error location l_{error} by the array-based system $\mathcal{S}_\mathcal{T}$ associated to P. This amounts to establish if (2.6) is $\mathcal{T}$-satisfiable for some $n \geq 0$. Assuming that the $\mathcal{T}$-satisfiability of formulæ of the form (2.6) is decidable, a possible way to solve the problem is to enumerate the instances of (2.6) for increasing values of n. When the error condition is reachable, the procedure terminates; otherwise, it diverges. It is in this latter case that new solutions have to be found to limit these divergence phenomena.

It is well-known (see, e.g., [Manna and Pnueli, 1995]) that one can show that $\mathcal{S}_\mathcal{T}$ is safe by providing a *safe inductive invariant* for it. A safe inductive invariant is, in our context, a formula $H(\mathbf{v})$ representing (an overapproximation of) the set of all possible configurations of $\mathcal{S}_\mathcal{T}$ such that $H(\mathbf{v}) \wedge pc = l_{\mathsf{error}}$ is not satisfiable. More formally,

Definition 2.4.2 (Invariants)**.** *A* safe inductive invariant *for $\mathcal{S}_\mathcal{T}$ is a formula $H(\mathbf{v})$ such that*

$$\begin{aligned} &(i)\ \mathcal{T} \models \forall \mathbf{v}.pc = l_{\mathsf{init}} \rightarrow H(\mathbf{v}) \\ &(ii)\ \mathcal{T} \models \forall \mathbf{v}, \mathbf{v}'.H(\mathbf{v}) \wedge T(\mathbf{v}, \mathbf{v}') \rightarrow H(\mathbf{v}') \\ &(iii)\ \mathcal{T} \models \forall \mathbf{v}.H(\mathbf{v}) \rightarrow pc \neq l_{\mathsf{error}} \end{aligned} \tag{2.7}$$

If $H(\mathbf{v})$ satisfies only (i) and (ii), it is said to be an inductive invariant *(but not safe).*

Example 2.4.1. Consider the procedure Running in Figure 2.1. The first loop of the procedure initializes the array b according to the content of the array a

such that, at the end of the loop, the following assertion holds:

$$\text{for every index } i \text{ in the range } 0...L,\ b[i] = \mathsf{true} \text{ iff } a[i] \geq 0. \tag{2.8}$$

The second loop of the procedure sets the Boolean flag f to false if a position in the array a contradicting (2.8) is found.

The program is clearly safe, i.e., the assertion after the second loop is satisfied for any execution of the procedure. To prove its safety, one has to provide a safe inductive invariant. A formula achieving this goal is the following:

$$\begin{aligned}
&pc = l_1 \rightarrow (\forall z_0.\, ((0 \leq z_0 \wedge z_0 < i) \rightarrow (a[z_0] \geq 0 \leftrightarrow b[z_0])) \quad \wedge \\
&pc = l_1 \rightarrow i \geq 0 \\
&pc = l_2 \rightarrow (\forall z_0.\, ((0 \leq z_0 \wedge z_0 < L) \rightarrow (a[z_0] \geq 0 \leftrightarrow b[z_0])) \quad \wedge \\
&pc = l_2 \rightarrow i \geq 0 \\
&pc = l_2 \rightarrow f \ \wedge \\
&pc = l_3 \rightarrow f \ \wedge \\
&pc \neq l_4
\end{aligned} \tag{2.9}$$

As we show in the next chapters, our approach can automatically generate the above formula.

Chapter 3

Lazy Abstraction with Interpolants for Arrays

This chapter presents an extension of the *Lazy Abstraction with Interpolants* (LAWI) [McMillan, 2006] framework suitable for the analysis of programs with arrays. The LAWI framework is one of the most efficient frameworks for the analysis of programs. It is an instance of the more general *CounterExample Guided Abstraction Refinement* (CEGAR) paradigm [Clarke et al., 2000]. The idea underlying CEGAR is to iteratively refine abstractions of a system by refuting abstract executions violating given properties that are not concretely reproducible. This iterative process is performed with respect to set of predicates P. This set is modified by exploiting Craig interpolants computed from unsatisfiable formulæ retrieved from the abstract spurious counterexamples, i.e., the abstract executions not representing any concrete undesired execution.

Verification tools based on the LAWI schema have been successfully applied to certain classes of programs, e.g., device drivers [Ball and Rajamani, 2002]. However, the annotations of such programs involve only simple properties about the data-flow with a limited interplay with the control-flow. When used to verify programs manipulating sophisticated data-structures such as arrays, CEGAR and Lazy Abstraction show some limitations. One of the most important reason for the the limited success of Lazy Abstraction on programs manipulating arrays is the fact that program annotations often require (universal) quantification, as shown in section 2.4.

The new framework proposed in this chapter enhance the standard LAWI approach enabling its application to programs requiring quantified invariants as follows. The solution presented in this chapter is developed in the Model Checking Modulo Theory approach [Ghilardi and Ranise, 2010a, Ghilardi and

Ranise, 2010b] in which verification is performed by a symbolic backward reachability procedure. Certain classes of formulæ represent sets of backward reachable states and fix-point checks are reduced to logical problems that SMT solvers are able to tackle, once extended with suitable quantifier instantiation techniques. The MCMT approach has been successfully exploited for the verification of parameterized (distributed) systems (see, e.g., [Ghilardi and Ranise, 2010a,Alberti et al., 2010a,Alberti et al., 2012d]) but it fails when applied to the verification of imperative programs because of the lack of suitable abstraction-refinement techniques. To overcome this problem, we extend the backward reachability procedure of MCMT with a carefully designed interpolation-based abstraction refinement technique capable to generate the quantified predicates required for the synthesis of the inductive invariants, needed to establish the safety of programs manipulating arrays. For this, we need to address the following technical challenges:

(i) Refinement must be able to deal with quantified formulæ, i.e. it is necessary to discover new predicates possibly containing quantifiers. Indeed, this is much more a difficult task than finding predicates that are equivalent to quantifier-free formulæ as it is the case in many Lazy Abstraction approaches focusing on scalar data structures (see, e.g., [Henzinger et al., 2004]). To understand the problem, consider the procedure Running in Figure 2.1 and recall that (2.8) is the invariant required for proving its safety. Refinement should be able to generate it as a single predicate, because of the universally quantified variable i; definitely a non-trivial task.

(ii) Satisfiability of formulæ representing (abstract) counter-examples must be decidable. This is key to be able to automatically detect when the abstract program requires to be refined. Unfortunately, the situation is complicated by the fact that interpolation-based refinement may introduce extra quantifiers in the new predicates because, as we discussed in section 2.2.1, the theory of arrays does not admit, in general, quantifier-free interpolation. As a consequence, refinement needs to be carefully controlled since the introduction of quantifiers may give rise to formulæ containing alternations of quantifiers. This easily leads to the undecidability of the satisfiability of the formulæ representing sets of backward reachable states.

(iii) The implementation of interpolation-based refinement procedures is delicate because the "quality" of the generated interpolants may generate too

many refinements, thereby degrading performances unacceptably, or even worse making the procedure diverging. This is so because a pair (A, B) of inconsistent formulæ may admit several (even infinitely many) interpolants and choosing the one that is "the best" with respect to refinement is an undecidable problem. To illustrate the problem, consider again the procedure Running in Figure 2.1. An interpolation-based refinement procedure may generate the sequence $b[0] \leftrightarrow a[0] \geq 0$, $b[1] \leftrightarrow a[1] \geq 0$, ... of infinitely many (quantifier-free) predicates. After each iteration of refinement, the conjunction of these predicates offers only an approximation of the quantified assertion (2.8) needed to prove the safety of Running and the Lazy Abstraction procedure diverges because of the infinite (increasingly precise) sequence of approximations. Heuristics (see, e.g., [Jhala and McMillan, 2006]) to tune the generation of interpolants and avoid divergence are crucial for efficient implementations.

Our solution tackles the aforementioned challenges by exploiting the following ideas. We will work with *flattened formulæ*, i.e., (recall Definitions2.2.1) formulæ where array variables are dereferenced only by existentially quantified variables. Thus, a formula of the kind $\phi(a[i], ...)$ (where i is a constant or more generally a term) is first rewritten as $\exists x\ (x = i \wedge \phi(a[x], ...))$. During consistency tests, the existentially quantified variable x is Skolemized away, so that consistency tests are made with quantifier-free formulæ. Interpolants search is performed at quantifier-free level. If the interpolant abstracts away the constant i from $x = i \wedge \phi(a[x], ...)$[1], when de-Skolemization reintroduces the variable x, this x will be a genuine existentially quantified variable. In fact, the negation of the resulting formula will be part of the universally quantified invariant we are looking for (recall that backward search produces, when successful, existentially quantified formulæ whose negations turn out to be invariants).

In summary, the contributions of this chapter are:

- a framework for abstraction-refinement with quantified predicates;
- a quantifier-free interpolation algorithm for a relevant class of formulæ with array variables.

[1]The next chapter will describe practical heuristics for achieving this goal.

3.1 Background

In the rest of the chapter we shall use the following notation an rely on the following conventions.

We fix a background theory of arrays $\mathcal{A}_I^E$, a multisorted theory with sort symbols $\texttt{INDEX}$, $\texttt{ELEM}_\ell$ and $\texttt{ARRAY}_\ell$, where $\texttt{ARRAY}_\ell$ is interpreted as the set of total functions from the interpretation of $\texttt{INDEX}$ to the interpretation of $\texttt{ELEM}_\ell$. The signature of $\mathcal{A}_I^E$ contains also the set of symbols $\{\_[\_]_\ell\}$, where $\_[\_]_\ell : \texttt{ARRAY}_\ell \times \texttt{INDEX} \rightarrow \texttt{ELEM}_\ell$ are the usual dereference operations for arrays, interpreted as function applications. The subscript ℓ will be omitted in the following for simplifying the notation. We will target mainly the verification of programs with arrays of integers. In this setting, $\mathcal{T}_I$ will be generally identified as an arithmetical theory, e.g., $\mathcal{LIA}$. $\texttt{INDEX}$ will be interpreted over $\mathbb{N}$. $\mathcal{T}_E$ will be a combination of theories, typically an enumerated data-type theory used to handle the control-flow of the program and an arithmetical theory ($\mathcal{LIA}$ or $\mathcal{IDL}$) for the content of the arrays. To keep the framework as general as possible, we prefer not to fix them, though. The only requirements are the decidability of the $SMT(\mathcal{T}_I)$- and $SMT(\mathcal{T}_E)$-problems and that $\mathcal{T}_I$ and $\mathcal{T}_E$ have quantifier-free interpolation.

Furthermore, given an array-based transition system $\langle \mathbf{v}, l_{\mathsf{init}}, l_{\mathsf{error}}, T \rangle$, we partition the tuple of variables $\mathbf{v}$ as follows

- the tuple $\mathbf{a} = a_0, \ldots, a_s$ contains variables of sort $\texttt{ARRAY}$;
- the tuple $\mathbf{c} = c_0, \ldots, c_t$ contains variables of sort $\texttt{INDEX}$ (called, *counters*);
- the tuple $\mathbf{d} = d_0, \ldots, d_u$ contains variables of sort $\texttt{ELEM}$ (called, *simple variables*). We assume d_0 to be the program counter variable.

In light of the additional constraints we just specified, guarded assignment in functional form becomes formulæ of the form

$$\exists \underline{k} \left(\begin{array}{l} \phi_L(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \wedge \\ \mathbf{a}' = \lambda j.\ G(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}, j, \mathbf{a}[j]) \wedge \\ \mathbf{c}' = H(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \wedge \\ \mathbf{d}' = K(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d})) \end{array} \right) \tag{3.1}$$

where $G = G_0, \ldots, G_s$, $H = H_0, \ldots, H_t$, $K = K_0, \ldots, K_u$ are tuples of case-defined functions.

Example 3.1.1. Consider again the formalization of the Running procedure given in section 2.3.1. The transitions $\tau_1, \tau_2, \tau_4, \tau_5, \tau_6$ and τ_7 are not matching

formula (3.1) since terms of the form $\mathbf{a}[\mathbf{c}]$ are not allowed. This is, however, without loss of generality. In fact, any formula $\psi(\cdots \mathbf{a}[\mathbf{c}] \cdots)$ containing such terms can be rewritten to $\exists \underline{x}(\underline{x} = \mathbf{c} \wedge \psi(\cdots \mathbf{a}[\underline{x}] \cdots))$ by using (fresh) existentially quantified variables $\underline{x}$ of sort `INDEX`:

$$\tau_1 := \begin{pmatrix} mov(l_1, l_1) \wedge i < L \wedge \exists x.(x = i \wedge a[x] \geq 0) \wedge \\ i' = i + 1 \wedge b' = \mathsf{store}(b, i, \mathsf{true}) \wedge id(a, f) \end{pmatrix}$$

$$\tau_2 := \begin{pmatrix} mov(l_1, l_1) \wedge i < L \wedge \exists x.(x = i \wedge a[x] < 0) \wedge \\ i' = i + 1 \wedge b' = \mathsf{store}(b, i, \mathsf{false}) \wedge id(a, f) \end{pmatrix}$$

$$\tau_4 := \begin{pmatrix} mov(l_2, l_2) \wedge i < L \wedge \exists x.(x = i \wedge b[x] \wedge a[x] < 0) \wedge \\ f' = \mathsf{false} \wedge i' = i + 1 \wedge id(a, b) \end{pmatrix}$$

$$\tau_5 := \begin{pmatrix} mov(l_2, l_2) \wedge i < L \wedge \exists x.(x = i \wedge \neg b[x] \wedge a[x] \geq 0) \wedge \\ f' = \mathsf{false} \wedge i' = i + 1 \wedge id(a, b) \end{pmatrix}$$

$$\tau_6 := mov(l_2, l_2) \wedge i < L \wedge \exists x.(x = i \wedge a[x] \geq 0 \wedge b[x]) \wedge i' = i + 1 \wedge id(a, b, f)$$

$$\tau_7 := mov(l_2, l_2) \wedge i < L \wedge \exists x.(x = i \wedge a[x] < 0 \wedge \neg b[x]) \wedge i' = i + 1 \wedge id(a, b, f) \ .$$

3.2 Unwindings for the safety of array-based transition systems

As introduced in section 2.4, naïve procedures for the establishment of the safety of a transition system $\mathcal{S}_\mathcal{T}$ diverge if $\mathcal{S}_\mathcal{T}$ is safe. A standard solution to avoid divergence is to compute the set of reachable states and check if a fix-point has been reached. The set of *forward* or *backward* reachable states is obtained by the repeated symbolic execution of transitions from the initial or the error location, respectively. For example, the symbolic execution of a transition τ from a set of states represented by a formula $K(\mathbf{v})$ amounts to the computation of the *pre-image* of $K(\mathbf{v})$ with respect to $\tau(\mathbf{v}, \mathbf{v}')$ as follows:

$$Pre(\tau, K) \ \equiv \ \exists \mathbf{v}'. \, (\tau(\mathbf{v}, \mathbf{v}') \wedge K(\mathbf{v}')) \, . \tag{3.2}$$

By taking the disjunction of the pre-images of $pc = l_{\mathsf{error}}$ with respect to all transitions, it is possible to compute the set of states from which l_{error} is reachable by applying just one transition. The reachability of the error location can be established with an iterative pre-image computation procedure, interleaved with checks for detecting fix-points or the presence of the initial location in the set of reachable states. Even when there is no sequence of transitions lead-

ing the system from the initial to the error location, it is possible to stop the procedure and conclude safety.

The problem with this procedure is that it is often impossible to compute fix-points for infinite state systems such as those associated to many programs. To alleviate this problem, an over-approximation of the set of reachable states is computed. This set has to be sufficiently coarse to permit the detection of a fix-point and sufficiently precise to show the safety of the analyzed system, if the case. In program verification it is a common practice to compute an over-approximation of the set of forward reachable states. In our case, given the backward reachability procedure, we consider the computation of an over-approximation of a backward reachable state-space. In this section, we show how it is possible to over-approximate the set of backward reachable states of an array-based system by using *labeled unwindings* [Henzinger et al., 2002].

3.2.1 Labeled unwindings for the safety of array-based systems

Preliminarily, we introduce some technical notions and notations. If ψ is a quantifier-free formula in which at most the index variables in $\underline{i}$ occur, we denote by $\psi^{\exists}$ its existential (index) closure, namely the formula $\exists \underline{i}\, \psi$. In addition, a $\forall^I$-formula is a formula of the kind $\forall \underline{i}.\phi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$, an $\exists^I$-formula is a one of the form $\exists \underline{i}.\phi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ and $\exists \mathbf{a} \exists \mathbf{c} \exists \mathbf{d} \exists \underline{i} \forall \underline{j}.\psi(\underline{i}, \underline{j}, \mathbf{a}[\underline{i}], \mathbf{a}[\underline{j}], \mathbf{c}, \mathbf{d})$ is a $\exists^{A,I}\forall^I$-sentence.

Definition 3.2.1. *A* labeled unwinding *of* $\mathcal{S}_{\mathcal{A}_I^E} = \langle \mathbf{v}; l_{\mathsf{init}}; l_{\mathsf{error}}; T \rangle$ *is a quadruple* (V, E, M_V, M_E)*, where* (V, E) *is a finite rooted tree (let* ε *be the root) and* M_V, M_E *are labeling functions for vertices and edges, respectively, such that:*

(*i*) *for every* $v \in V$*, if* $v = \varepsilon$*, then* $M_V(\varepsilon)$ *is* $pc = l_{\mathsf{error}}$*; otherwise (i.e.* $v \neq \varepsilon$*),* $M_V(v)$ *is a quantifier-free formula of the kind* $\psi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ *such that* $M_V(v) \models_{\mathcal{A}_I^E} pc = l$ *for some location* l*;*

(*ii*) *for every* $(v, w) \in E$*,* $M_E(v, w)$ *is the matrix of some* $\tau \in T$*; the proper variables of* τ *do not occur in* $M_V(w)$*; moreover, we have that* $M_V(w) \models_{\mathcal{A}_I^E} pc = trg(\tau)$*, that* $M_V(v) \models_{\mathcal{A}_I^E} pc = src(\tau)$*, and that*

$$M_E(v, w)(\mathbf{v}, \mathbf{v}') \wedge M_V(w)(\mathbf{v}') \models_{\mathcal{A}_I^E} M_V(v)(\mathbf{v}); \tag{3.3}$$

(*iii*) *for each* $\tau \in \{\tau_h(\mathbf{v}, \mathbf{v}')\}_h$ *and every non-leaf vertex* $w \in V$ *such that* $M_V(w) \models_{\mathcal{A}_I^E} pc = trg(\tau)$*, there exist* $v \in V$ *and* $(v, w) \in E$ *such that* $M_E(v, w)$ *is the matrix of* τ*.*

The intuition underlying this definition is that a vertex v in a labeled unwinding corresponds to a program location (i) and an edge (v, w) to the execution of a transition, whose source and target locations match with those of v and w, respectively (ii) and (iii). A closer look at condition (3.3) allows us to show how the set of backward reachable states obtained by repeatedly computing pre-images (3.2) can be over-approximated by the the formulæ attached to the vertices of a labeled unwinding. For this, we show that $M_V(v)^\exists$, i.e. the set of states associated to vertex v, *overapproximates* the set of states in the pre-image of $M_V(w)^\exists$ with respect to a transition τ.

Lemma 3.2.1. *Let $(u, w) \in E$ be an arc in a labeled unwinding (V, E, M_E, M_V); we have*

$$Pre(\tau, M_V(w)^\exists) \models_{\mathcal{A}_I^E} M_V(v)^\exists$$

where τ is the guarded assignment in functional form whose matrix is $M_E(v, w)$.

Proof. If we introduce existential quantifiers in both members of (3.3), we get

$$\exists \mathbf{v}'(M_E(v, w)(\mathbf{v}, \mathbf{v}') \wedge M_V(w)(\mathbf{v}'))^\exists \models_{\mathcal{A}_I^E} M_V(v)(\mathbf{v})^\exists;$$

taking into consideration that the proper variables of τ are the only index variables occurring free in the matrix of τ and that such proper variables do not occur in $M_V(w)$, we can move inside index quantifiers and get

$$\exists \mathbf{v}'(M_E(v, w)(\mathbf{v}, \mathbf{v}')^\exists \wedge M_V(w)(\mathbf{v}')^\exists) \models_{\mathcal{A}_I^E} M_V(v)(\mathbf{v})^\exists;$$

which is the claim because $M_E(v, w)(\mathbf{v}, \mathbf{v}')^\exists$ is $\tau(\mathbf{v}, \mathbf{v}')$. □

From this, it is clear that the disjunction of the existential index closure of the formulæ labeling the vertices of an unwinding is an over-approximation of the set of backward reachable states. As discussed above, the over-approximation is useful only when it allows us to prove safety when this is the case, i.e. when the approximation is not too coarse. This is equivalent to saying that the negation of the formula representing the over-approximated set of (backward) reachable states is an invariant of the system. We now characterize the conditions (see Definition 3.2.2 below) under which this is possible.

A set C of vertexes in a labeled unwinding (V, E, M_V, M_E) *covers* a vertex $v \in V$ iff

$$M_V(v)^\exists \models_{\mathcal{A}_I^E} \bigvee_{w \in C} M_V(w)^\exists. \tag{3.4}$$

Definition 3.2.2. *The labeled unwinding* (V, E, M_V, M_E) *is* safe *iff for all* $v \in V$ *we have that if* $M_V(v) \models pc = l_{\mathsf{init}}$*, then* $M_V(v)$ *is* $\mathcal{A}_I^E$*-unsatisfiable. It is* complete *iff there exists a* covering*, i.e., a set of non-leaf vertexes* C *containing* ε *and such that for every* $v \in C$ *and* $(v', v) \in E$*, it happens that* C *covers* v'*.*

The reader familiar with [McMillan, 2006] may have noticed that our notion of covering involves a set of vertexes rather than a single one as in [McMillan, 2006]. Indeed, an efficient implementation of our notion is crucial for efficiency and is discussed in chapter 4. Here, we focus on abstract definitions which allow us to prove that safe and complete labeled unwindings can be seen as safety certificates for array-based systems.

Theorem 3.2.1. *If there exists a safe and complete labeled unwinding of* $\mathcal{S}_{\mathcal{A}_I^E} = \langle \mathbf{v}; l_{\mathsf{init}}; l_{\mathsf{error}}; T \rangle$*, then* $\mathcal{S}$ *is safe.*

Proof. Let (V, E, M_V, M_E) be a safe and complete labeled unwinding of $\mathcal{S}$ with covering C. We show that $\bigvee_{w \in C} M_V(w)^{\exists}$, which is a disjunction of $\exists^I$-formulæ having the variables in $\mathbf{v} = \mathbf{a}, \mathbf{c}, \mathbf{d}$ as free variables, overapproximates the set of the system states that can reach the error location. More formally, we show that for every n the formula

$$T(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \ \wedge \ \cdots \ \wedge \ T(\mathbf{v}^{(1)}, \mathbf{v}^{(0)}) \ \wedge \ pc^{(0)} = l_{\mathsf{error}}$$

$\mathcal{A}_I^E$-entails the formula $\bigvee_{w \in C} M_V(w)^{\exists}(\mathbf{v}^{(n)})$. This implies also that the formula (2.6) cannot be satisfiable, because (V, E, M_V, M_E) is safe. Indeed, if (2.6) is satisfiable and the claim holds, this means that $pc^{(n)} = l_{\mathsf{init}} \wedge \bigvee_{w \in C} M_V(w)^{\exists}(\mathbf{v}^{(n)})$ is satisfiable, which can only be if some of the $M_V(w)$ is consistent and $\mathcal{A}_I^E$-entails $pc = l_{\mathsf{init}}$, i.e. if (V, E, M_V, M_E) is unsafe.

The proof of the statement is by induction on n. The case $n = 0$ is trivial because $\varepsilon \in C$ is labeled $pc = l_{\mathsf{error}}$; so suppose $n > 0$. By induction hypothesis, we need to show that

$$\bigvee_h \tau_h(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \wedge \bigvee_{w \in C} M_V(w)^{\exists}(\mathbf{v}^{(n-1)}) \models_{\mathcal{A}_I^E} \bigvee_{w \in C} M_V(w)^{\exists}(\mathbf{v}^{(n)})$$

i.e. that for each $\tau \in \{\tau_h\}_h$ and $v \in C$ we have

$$\tau(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \wedge M_V(v)^{\exists}(\mathbf{v}^{(n-1)}) \models_{\mathcal{A}_I^E} \bigvee_{w \in C} M_V(w)^{\exists}(\mathbf{v}^{(n)}).$$

By the definition of a labeled unwinding, either there is a location mismatch and $\tau(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \wedge M_V(v)^{\exists}(\mathbf{v}^{(n-1)})$ is inconsistent, or according to Defini-

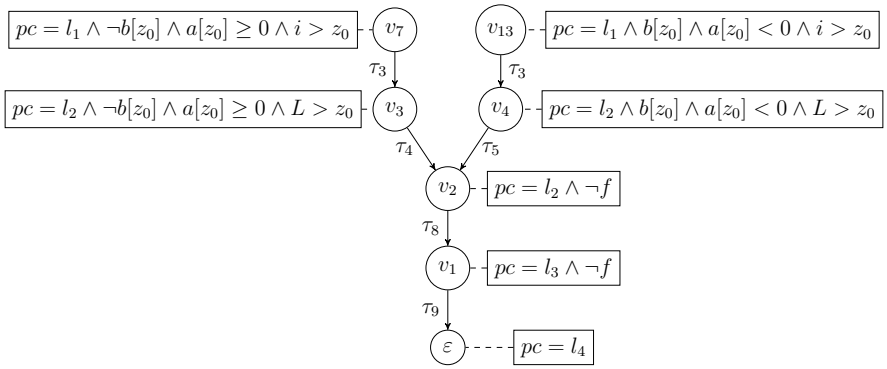


Figure 3.1. Covering associated with a labeled unwinding proving the safety of the Running procedure (the entire labeled unwinding has 77 vertices and 188 edges). The variable z_0 has sort `INDEX` and is introduced during backward reachability.

tion 3.2.1(iii) there must be a vertex v' with an edge (v', v) labeled by the matrix of τ in the tree (V, E) (this is because coverings do not contain leaves, hence v is not a leaf). We can now derive our claim from the definition of a covering and the fact that $\tau(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \wedge M_V(v)^{\exists}(\mathbf{v}^{(n-1)})$ $\mathcal{A}_I^E$-entails the formula $M_V(v')^{\exists}(\mathbf{v}^{(n)})$ by Lemma 3.2.1. □

As a final remark, we point out that safe and complete labeled unwindings are *quantified* safety certificates for array-based systems. To see why, consider the covering C associated with a safe and complete labeled unwinding. Then, a safe inductive invariant for the array based transition system is represented by the formula

$$\bigwedge_{w \in C} \neg \left(M_V(w)^{\exists}(\mathbf{v}) \right) . \tag{3.5}$$

Example 3.2.1. Consider again the transition system representing the Running procedure. Our framework can generate a safe and complete labeled unwinding for such transition system. The covering associated with this labeled unwinding is depicted in Figure 3.1, and represents the following invariant:

$$\begin{aligned}
&pc = l_1 \rightarrow (\forall z_0. \, ((0 \leq z_0 \wedge z_0 < i) \rightarrow (a[z_0] \geq 0 \leftrightarrow b[z_0])) \quad \wedge \\
&pc = l_2 \rightarrow (\forall z_0. \, ((0 \leq z_0 \wedge z_0 < L) \rightarrow (a[z_0] \geq 0 \leftrightarrow b[z_0])) \quad \wedge \\
&pc = l_2 \rightarrow f \wedge \\
&pc = l_3 \rightarrow f \wedge \\
&pc \neq l_4
\end{aligned}$$

If compared with the invariant given in section (2.4), i.e., the formula (2.9),

the invariant reported here is missing two conjuncts, i.e., $pc = l_1 \rightarrow i \geq 0$ and $pc = l_2 \rightarrow i \geq 0$. This is because here we assume that `INDEX` is interpreted over $\mathbb{N}$. Notably, the framework we will describe in chapter 8 is able to generate automatically these two invariants by means of an abstract interpreter based on the polyhedra abstract domain.

3.2.2 On checking the safety and completeness of labeled unwindings

Theorem 3.2.1 states that the safety of an array-based system can be established by checking if there exists a labeled unwinding that is safe and complete. A procedure for searching such an unwinding will be described in the next section. For the moment, assume that a candidate labeled unwinding has been found and consider the problem to check if it is safe and complete.

It is easy to see that the safety check can be reduced to the $\mathcal{A}_I^E$-satisfiability of a quantifier-free formula. In fact, the formula $M_V(v)$ associated to a vertex v in a labeled unwinding is quantifier-free by Definition 3.2.1.(i). According to Definition 3.2.2, testing safety amounts to checking unsatisfiability of quantifier-free formulæ. This problem is decidable, as proven by Lemma 2.2.2, provided that both the $SMT(\mathcal{T}_I)$ and $SMT(\mathcal{T}_E)$ problems are decidable; recall that this has been assumed in section 3.1.

Checking the completeness of a labeled unwinding is more involved. According to Definition 3.2.2, this requires to guess a sub-set C of the set of vertexes in the unwinding and check if C covers v', for every $v \in C$ and $(v', v) \in E$. In turn, by refutation from (3.4), this may be reduced to repeatedly check the $\mathcal{A}_I^E$-unsatisfiability of $\exists^{A,I}\forall^I$-sentences, i.e. formulæ of the form

$$\exists \mathbf{a}\, \exists \mathbf{c}\, \exists \mathbf{d}\, \exists \underline{i}\, \forall \underline{j}.\ \psi(\underline{i}, \underline{j}, \mathbf{a}[\underline{i}], \mathbf{a}[\underline{j}], \mathbf{c}, \mathbf{d})\,, \tag{3.6}$$

where $\underline{i}, \underline{j}, \mathbf{c}$ are of sort `INDEX`, $\mathbf{a}$ are of sort `ARRAY` and $\mathbf{d}$ of sort `ELEM`. Unfortunately, the $\mathcal{A}_I^E$-satisfiability of these sentences is (in general) undecidable [Ghilardi and Ranise, 2010a]. The problem is the handling of the universally quantified variables of $\underline{j}$ that occur in (3.6) since all the other existentially quantified variables in $\mathbf{a}$, $\mathbf{c}$, $\mathbf{d}$, and $\underline{i}$ can be regarded as Skolem constants. To alleviate the problem, an idea is to design an incomplete instantiation procedure for the variables in $\underline{j}$ to obtain a conjunction of quantifier-free formulæ whose $\mathcal{A}_I^E$-satisfiability is decidable by Lemma 2.2.2. Our *default instantiation procedure* computes the set Σ of all possible substitutions mapping the variables in $\underline{j}$ into $\underline{i} \cup \mathbf{c}$. Our *default satisfiability procedure* uses the default instantiation

procedure to check the $\mathcal{A}_I^E$-unsatisfiability of the formula

$$\bigwedge_{\sigma\in\Sigma} \psi(\underline{i}, \underline{j}\sigma, \mathbf{a}[\underline{i}], \mathbf{a}[\underline{j}], \mathbf{c}, \mathbf{d})\,. \tag{3.7}$$

It returns the $\mathcal{A}_I^E$-unsatisfiability of (3.6) when (3.7) is so and returns "unknown" when (3.7) is $\mathcal{A}_I^E$-satisfiable.

In other words, the default satisfiability procedure is sound but incomplete for checking the $\mathcal{A}_I^E$-satisfiability of $\exists^{A,I}\forall^I$-sentences. In section 3.4, we show that the adoption of such a procedure allows us to use labeled unwindings as safety certificates. To clarify that the notion of completeness for labeled unwindings is relative to the incomplete algorithm used to check the completeness of coverings, we introduce the following notion.

Definition 3.2.3. *The labeled unwinding* (V, E, M_V, M_E) *is* recognized to be complete *iff there exists a set of non-leaf vertexes* C *(called a 'recognized covering' or simply a 'covering' for the sake of simplicity) containing* ε *and such that for every* $v \in C$ *and* $(v', v) \in E$*, it happens that the relation* (3.4) *is verified to hold by using the default satisfiability procedure for* $\mathcal{A}_I^E$*-satisfiability of* $\exists^{A,I}\forall^I$*-sentences.*

In section 3.4, we will identify sufficient conditions under which the default instantiation procedure allows us to build a decision procedure for the $\mathcal{A}_I^E$-satisfiability problem of $\exists^{A,I}\forall^I$-sentences. We will also see that the same conditions guarantee the termination of the procedure described in the next section that finds a safe and complete labeled unwinding.

In chapter 4, we will describe heuristics to reduce the number of possible instances that must be considered by the default instantiation procedure to improve performance. The experiments described in section 4.2 show the efficiency of the default satisfiability procedure described above.

3.3 Lazy abstraction with interpolation-based refinement for arrays

We now describe how to construct labeled unwindings and how this process is interleaved with the checks for safety and completeness described in section 3.2.2. Similarly to [McMillan, 2006], we design a (possibly non-terminating) procedure UNWIND, that – given an array-based system $\mathcal{S}_{\mathcal{A}_I^E}$ – computes a sequence of (increasingly larger) labeled unwindings. The initial labeled unwinding of $\mathcal{S}_{\mathcal{A}_I^E}$ is the tree containing just the root labeled by $pc = l_{\mathsf{error}}$.

UNWIND uses two sub-procedures: EXPAND builds the labeled unwinding and REFINE refines labeled unwindings by eliminating spurious unsafe traces via interpolants. When REFINE is applicable but fails, $\mathcal{S}_{\mathcal{A}_I^E}$ is unsafe. If none of the two procedures applies, then the current labeled unwinding is safe and complete: $\mathcal{S}_{\mathcal{A}_I^E}$ is safe by Theorem 3.2.1.

As we will see below, a crucial advantage of our approach is that REFINE *needs to compute only quantifier-free interpolants (in a restricted form) to refine spurious unsafe traces, despite the fact that quantified formulæ are used to represent sets of states and transitions.* Technically, this is possible because formulæ describing potentially unsafe traces can be transformed to equisatisfiable quantifier-free formulæ by a partial instantiation procedure (see section 3.3.2 below for details).

In the following, we give a non-deterministic version of UNWIND: the two procedures EXPAND and UNWIND can be non-deterministically applied to a labeled unwinding to obtain a new one, whenever this is possible according to their applicability conditions (described below). The implementation strategies of UNWIND will be described in section 4.1.

3.3.1 The two sub-procedures of UNWIND

Let (V, E, M_V, M_E) be the current labeled unwinding of $\mathcal{S}_{\mathcal{A}_I^E}$. From now on, we assume that *the initial location is not a target location, the error location is not a source location*, and that initial and error locations are *the only locations that are not both a source and a target location.*

EXPAND. The applicability condition is that (V, E, M_V, M_E) is not recognized to be complete (recall Definition 3.2.3) and there exists a leaf vertex v whose location is such that $M_V(v) \not\models_{\mathcal{A}_I^E} pc = l_{\mathsf{init}}$.

The effects of applying this procedure are the following: for each transition $\tau \in T$ whose target is l, a new leaf w_τ, labeled by $pc = src(\tau)$, is added together with a new edge (w_τ, v), labeled by τ, to the current unwinding.

REFINE. The applicability condition is that (V, E, M_V, M_E) is not recognized to be complete (recall Definition 3.2.3) and there exists a vertex $v \in V$ whose location is l_{init} and it is such that $M_V(v)$ is $\mathcal{A}_I^E$-satisfiable.

In the current labeled unwinding, consider the path $v = v_0 \to v_1 \to \cdots \to v_m = \varepsilon$ from v to the root and let $\tau_1, \ldots, \tau_m$ be the transitions

labeling the edges from left to right; the set of these transitions is called a *counterexample*. If

$$\tau_1(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \cdots \wedge \tau_m(\mathbf{v}^{(m-1)}, \mathbf{v}^{(m)}) \tag{3.8}$$

is $\mathcal{A}_I^E$-satisfiable then the counterexample is said to be *feasible*, the procedure fails, and reports the unsafety of $\mathcal{S}_{\mathcal{A}_I^E}$. Otherwise, the counterexample is said to be *infeasible* and the effect of applying the procedure is to strengthen the labels of the counterexample vertices by using the interpolants retrieved from the unsatisfiability of (3.8).

The mechanization of the applicability conditions for both sub-procedures have been discussed in section 3.2.2. This means that enough details for the mechanization of Expand are already available. This is not the case for Refine because it is unclear how to check the $\mathcal{A}_I^E$-satisfiability of formulæ of the form (3.8)—this is crucial to establish the feasibility or infeasibility of a counterexample—and we do not know how to compute interpolants and how to use them in order to "strengthen the labels in the counterexample."

The feasibility of counterexamples is discussed in section 3.3.2, the computation of (quantifier-free) interpolants in section 3.3.4, and their use in refining (infeasible) counterexamples in section 3.3.3.

3.3.2 Checking the feasibility of counterexamples

We describe a decision procedure for checking the $\mathcal{A}_I^E$-satisfiability of formulæ of the form (3.8), thereby enabling to check the feasibility of counterexamples in Refine. The idea underlying the procedure is to instantiate the variables bound by the λ-abstraction in the updates of the transitions occurring in (3.8) with finitely many constants and then check the resulting quantifier-free formula for $\mathcal{A}_I^E$-satisfiability. The fact that only finitely many instances are sufficient is shown by the following observations.

By recalling (3.1), rewrite (3.8) to

$$\bigwedge_{k=1}^{m} \exists \underline{i}_k \left[\begin{array}{r} \phi_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \wedge \\ \mathbf{a}^{(k)} = \lambda j.\ G_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}, j, \mathbf{a}^{(k-1)}[j]) \wedge \\ \mathbf{c}^{(k)} = H_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \wedge \\ \mathbf{d}^{(k)} = K_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \end{array}\right] \tag{3.9}$$

which, by Skolemizing existentially quantified variables, can be further rewrit-

ten to the equi-satisfiable formula (here and in the following, by abuse of notation, we consider the variables in $\underline{i}_k$ as Skolem constants):

$$\bigwedge_{k=1}^{m} \left[\begin{array}{r} \phi_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \wedge \\ \mathbf{a}^{(k)} = \lambda j.\, G_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}, j, \mathbf{a}^{(k-1)}[j]) \wedge \\ \mathbf{c}^{(k)} = H_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \wedge \\ \mathbf{d}^{(k)} = K_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \end{array} \right] . \tag{3.10}$$

Now, observe that $\mathbf{a}^{(k)} = \lambda j\, G_k(\dots)$ is equivalent to $\forall j.\ \mathbf{a}^{(k)}[j] = G_k(\dots j \dots)$ and *instantiate the variable j with the Skolem constants in $\underline{i}_{k+1}, ..., \underline{i}_m$* to derive

$$\bigwedge_{k=1}^{m} \left[\begin{array}{r} \phi_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \wedge \\ \bigwedge_{j \in \underline{i}_{k+1}, \dots, \underline{i}_m} \mathbf{a}^{(k)}[j] = G_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}, j, \mathbf{a}^{(k-1)}[j]) \wedge \\ \mathbf{c}^{(k)} = H_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \wedge \\ \mathbf{d}^{(k)} = K_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \end{array} \right] \tag{3.11}$$

Lemma 3.3.1. *Formulæ* (3.10) *and* (3.11) *are $\mathcal{A}_I^E$-equisatisfiable.*

Proof. Indeed, (3.10) $\mathcal{A}_I^E$-entails (3.11). Vice-versa, suppose we are given an $\mathcal{A}_I^E$-model $\mathcal{M}$ and a satisfying assignment $\mathtt{s}$ for (3.11), our goal is to produce a satisfying assignment $\tilde{\mathtt{s}}$ for (3.10) based on the same $\mathcal{A}_I^E$-model $\mathcal{M}$. For simplicity, let us call $\underline{i}_1, \dots, \underline{i}_m, \mathbf{v}^{(0)}, \dots, \mathbf{v}^{(m)}$ the elements from the support of $\mathcal{M}$ assigned by $\mathtt{s}$ to the variables $\underline{i}_1, \dots, \underline{i}_m, \mathbf{v}^{(0)}, \dots, \mathbf{v}^{(m)}$ occurring free in (3.10) and (3.11). The assignment $\tilde{\mathtt{s}}$ will change only the values assigned to $\mathbf{v}^{(1)}, \dots, \mathbf{v}^{(m)}$ (notice that $\mathbf{v}^{(0)}$ is left unchanged). We define $\tilde{\mathtt{s}}(\mathbf{v}^k)$ for $k > 0$ inductively as follows:

$$\begin{array}{ll} \tilde{\mathtt{s}}(\mathbf{a}^{(k)}) & = \lambda j\, G_k(\underline{i}_k, \tilde{\mathtt{s}}(\mathbf{a}^{(k-1)})[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}, j, \tilde{\mathtt{s}}(\mathbf{a}^{(k-1)})[j]) \\ \tilde{\mathtt{s}}(\mathbf{c}^{(k)}) & = H_k(\underline{i}_k, \tilde{\mathtt{s}}(\mathbf{a}^{(k-1)})[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \\ \tilde{\mathtt{s}}(\mathbf{d}^{(k)}) & = K_k(\underline{i}_k, \tilde{\mathtt{s}}(\mathbf{a}^{(k-1)})[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \end{array}$$

To show that (3.10) holds under $\tilde{\mathtt{s}}$, a simple induction on k $(= 1, \dots, m)$ is sufficient to check that $\tilde{\mathtt{s}}(\mathbf{c}^{(k-1)}) = \mathbf{c}^{(k-1)}$, $\tilde{\mathtt{s}}(\mathbf{d}^{(k-1)}) = \mathbf{d}^{(k-1)}$ and $\tilde{\mathtt{s}}(\mathbf{a}^{(k-1)})[j] = \mathbf{a}^{(k-1)}[j]$ for all $j \in \underline{i}_k \cup \dots \cup \underline{i}_m$. As a consequence of this, the formulæ ϕ_k's still hold under $\tilde{\mathtt{s}}$ and the remaining conjuncts of (3.10) hold by construction. □

An easy corollary of Lemmas 3.3.1 and 2.2.2 is the following result.

Lemma 3.3.2. *The $\mathcal{A}_I^E$-satisfiability of formulæ of the form* (3.8) *is decidable.*

This means that we can check the feasibility of counterexamples under the assumption that the SMT problems of the theory $\mathcal{T}_I$ over indexes and the theory $\mathcal{T}_E$ over elements are decidable (recall that this has been assumed in section 3.1). A by-product of this result is the decidability of the bounded model checking problem (formally defined below) for array-based systems.

Let $\mathcal{S}_{\mathcal{A}_I^E} = \langle \mathbf{v}; l_{\mathsf{init}}; l_{\mathsf{error}}; T \rangle$ and recall the formula (2.6), i.e.

$$pc^{(n)} = l_{\mathsf{init}} \ \wedge \ \bigwedge_{i=1}^{n} T(\mathbf{v}^{(i)}, \mathbf{v}^{(i-1)}) \ \wedge \ pc^{(0)} = l_{\mathsf{error}} \tag{2.6}$$

When $n \geq 0$ is known, we say that the *bounded model checking problem* for $\mathcal{S}_{\mathcal{A}_I^E}$ consists of checking the $\mathcal{A}_I^E$-satisfiability of the formula above for the given value of n. We now show that Lemmas 3.3.1 and 2.2.2 also imply the decidability of this problem.

First of all, observe that, by applying standard distributive laws[2] and renaming of variables (the variable $\mathbf{v}^{(k)}$ is renamed to $\mathbf{v}^{(n-k)}$, so $\mathbf{v}^{(n)}$ is renamed to $\mathbf{v}^{(0)}$, $\mathbf{v}^{(n-1)}$ to $\mathbf{v}^{(1)}$, ..., $\mathbf{v}^{(1)}$ to $\mathbf{v}^{(n-1)}$, and $\mathbf{v}^{(0)}$ to $\mathbf{v}^{(n)}$), the formula above can be rewritten to a disjunction of formulæ of the form

$$pc^{(0)} = l_{\mathsf{init}} \wedge \tau_{h_1}(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \cdots \wedge \tau_{h_n}(\mathbf{v}^{(n-1)}, \mathbf{v}^{(n)}) \wedge pc^{(n)} = l_{\mathsf{error}} \,, \tag{3.12}$$

where h_j ranges over the same set of indexes of the transitions in $\mathcal{S}_{\mathcal{A}_I^E}$ and $j = 1, ..., n$. Now, observe that $\tau_{h_1}(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \cdots \wedge \tau_{h_n}(\mathbf{v}^{(n-1)}, \mathbf{v}^{(n)})$ has the same form of (3.8) and, by Lemma 3.3.1, it is $\mathcal{A}_I^E$-equisatisfiable to a quantifier-free formula ϕ of the form (3.11). The decidability of (2.6) is now obvious because every transition formula $\tau_h(\mathbf{v}, \mathbf{v}')$ entails $pc = src(\tau_h) \wedge pc' = trg(\tau_h)$ (recall the definition of a guarded assignment in functional form from section 2.3) and, "modulo" $\mathcal{A}_I^E$ formulæ of the form $l_1 = l_2$, are unsatisfiable when locations l_1 and l_2 are distinct. Thus (3.12) is either trivially unsatisfiable (in case of the locations are different) or equisatisfiable to ϕ. From this observation follows the following result.

Theorem 3.3.1. *The bounded model checking problem for array-based systems is decidable.*

[2]Recall that we assumed $T(\mathbf{v}, \mathbf{v}) \equiv \bigvee_{\tau \in T} \tau(\mathbf{v}, \mathbf{v}')$.

3.3.3 Refining counterexamples with interpolants

Assume that REFINE has detected that the infeasibility of the counterexample associated with the path $v_0 \xrightarrow{\tau_1} v_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_m} v_m = \varepsilon$ as shown in section 3.3.2, i.e. by the checking the $\mathcal{A}_I^E$-unsatisfiability of the formula $\tau_1 \wedge \cdots \wedge \tau_m$ of the form (3.8). At this point, REFINE needs to refine the counterexample. Following [McMillan, 2006], this is done by computing *path interpolants* that are conjoined to the labels of the vertices of the path under consideration to strengthen them. This is detailed in the following by assuming the availability of a procedure capable to compute interpolants for quantifier-free formulæ (the description of such a procedure is postponed to section 3.3.4).

Let us consider an $\mathcal{A}_I^E$-unsatisfiable formula of the form (3.8). By Lemma 3.3.1, this formula is $\mathcal{A}_I^E$-equisatisfiable to a quantifier-free formula of the form (3.11). This implies that also (3.11) is $\mathcal{A}_I^E$-unsatisfiable. Let us abbreviate the k-th conjunct in (3.11) as

$$\tilde{\tau}_k(\underline{i}_k, \ldots, \underline{i}_m, \mathbf{a}^{(k-1)}[\underline{i}_k], \ldots, \mathbf{a}^{(k-1)}[\underline{i}_m], \mathbf{a}^{(k)}[\underline{i}_{k+1}], \ldots, \mathbf{a}^{(k)}[\underline{i}_m], tc^{(k-1)}, \mathbf{c}^{(k)}, \mathbf{d}^{(k-1)}, \mathbf{d}^{(k)}) \,. \tag{3.13}$$

Thus, (3.11) can be written as $\tilde{\tau}_1 \wedge \cdots \wedge \tilde{\tau}_m$. Now, let

$$\psi_k(\underline{i}_{k+1}, \ldots, \underline{i}_m, \mathbf{a}[\underline{i}_{k+1}], \ldots, \mathbf{a}[\underline{i}_m], \mathbf{c}, \mathbf{d}) \tag{3.14}$$

be one of the quantifier-free interpolants (for $k = 1, ..., m$)—computed by repeatedly invoking the available interpolation procedure on the $\mathcal{A}_I^E$-unsatisfiable formula (3.11) from right-to-left. The ψ_k's are such that

$$\psi_0 \equiv \bot \tag{3.15}$$

$$\begin{aligned} \psi_k(\underline{i}_{k+1}, \ldots, \underline{i}_m, \mathbf{a}^{(k)}[\underline{i}_{k+1}], \ldots, \mathbf{a}^{(k)}[\underline{i}_m], \mathbf{c}^{(k)}, \mathbf{d}^{(k)}) \;\wedge\; \tilde{\tau}_k \quad &\models_{\mathcal{A}_I^E} \\ \psi_{k-1}(\underline{i}_k, \ldots, \underline{i}_m, \mathbf{a}^{(k-1)}[\underline{i}_k], \ldots, \mathbf{a}^{(k-1)}[\underline{i}_m], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) & \end{aligned} \tag{3.16}$$

$$\psi_m \equiv \top \tag{3.17}$$

Once these interpolants are computed, REFINE updates the label of v_k, for $k = 0, \ldots, m-1$, in the path $v_0 \xrightarrow{\tau_1} v_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_m} v_m = \varepsilon$ as follows:

$$M_V(v_k) \;:=\; M_V(v_k) \wedge \psi_k(\underline{i}_{k+1}, \ldots, \underline{i}_m, \mathbf{a}[\underline{i}_k], \ldots, \mathbf{a}[\underline{i}_m], \mathbf{c}, \mathbf{d}). \tag{3.18}$$

Since the matrix of τ_k $\mathcal{A}_I^E$-entails $\tilde{\tau}_k$, condition (3.3) of Definition 3.2.1.(*ii*) of labeled unwinding (see section 3.2.1) is preserved and the vertex v_0 is now labeled by an $\mathcal{A}_I^E$-unsatisfiable formula.

Example 3.3.1. Consider again the procedure Running in Figure 2.1, and

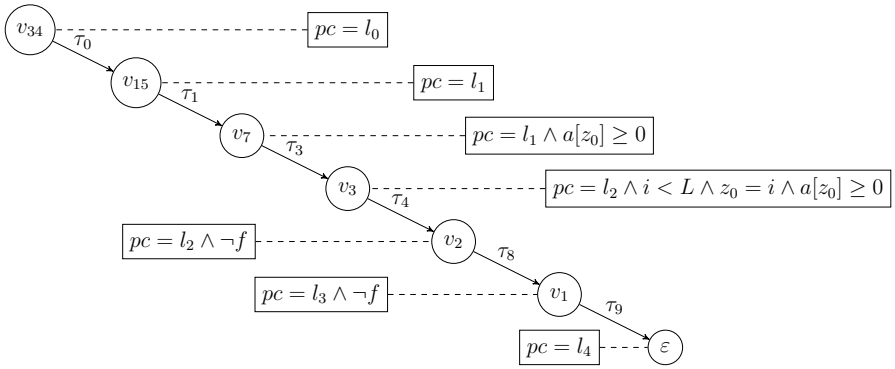


Figure 3.2. A candidate counterexample generated by the EXPAND procedure.

suppose that EXPAND produced a labeled unwinding containing the path depicted in Figure 3.2. This path triggers the execution of REFINE. This procedure checks whether the formula associated with this counterexample is $\mathcal{A}_I^E$-satisfiable exploiting the decision procedure described in section 3.3.2. The quantifier-free formula resulting after the selective instantiation is shown below:[3]

$$mov(l_0, l_1, 1) \wedge i^{(1)} = 0 \wedge id(f, a[z_0], b[z_0], 1)$$
$$mov(l_1, l_1, 2) \wedge i^{(2)} = i^{(1)} + 1 \wedge z_0 = i^{(1)} \wedge 0 \leq a^{(1)}[z_0] \wedge b^{(2)}[z_0] \wedge i^{(1)} < L \wedge id(f, a[z_0], 2)$$
$$mov(l_1, l_2, 3) \wedge i^{(2)} \geq L \wedge i^{(3)} = 0 \wedge f^{(3)} \wedge id(a[z_0], b[z_0], 3)$$
$$mov(l_2, l_2, 4) \wedge 0 \leq a^{(3)}[z_0] \wedge \neg b^{(3)}[z_0] \wedge i^{(3)} < L \wedge z_0 = i^{(3)} \wedge i^{(4)} = i^{(3)} + 1 \wedge \neg f^{(4)}$$
$$mov(l_2, l_3, 5) \wedge i^{(4)} \geq L \wedge id(f, i, 5)$$
$$mov(l_3, l_4, 6) \wedge \neg f^{(5)} \wedge id(f, i, 6)$$

This formula is $\mathcal{A}_I^E$-unsatisfiable. REFINE computes, therefore, a set of interpolants. For this counterexample, the computed interpolants are $\psi_0 := \bot, \psi_1 := \bot, \psi_2 := \neg b[z_0], \psi_3 := \neg b[z_0], \psi_4 := \top, \psi_5 := \top, \psi_6 := \top$. These formulæ are conjoined to the labels of the corresponding vertices in the path shown in in Figure 3.2 that is refined to the one depicted in Figure 3.3.

[3]For the sake of readability, $mov(l_i, l_j, k)$ stands for $pc^{(k-1)} = l_i \wedge pc^{(k)} = l_j$ and $id(t_1, \ldots, t_n; k)$ for $t_1^{(k)} = t_1^{(k-1)} \wedge \ldots \wedge t_n^{(k)} = t_n^{(k-1)}$. The Skolem variables introduced by REFINE are denoted by z_j for $j \geq 0$.

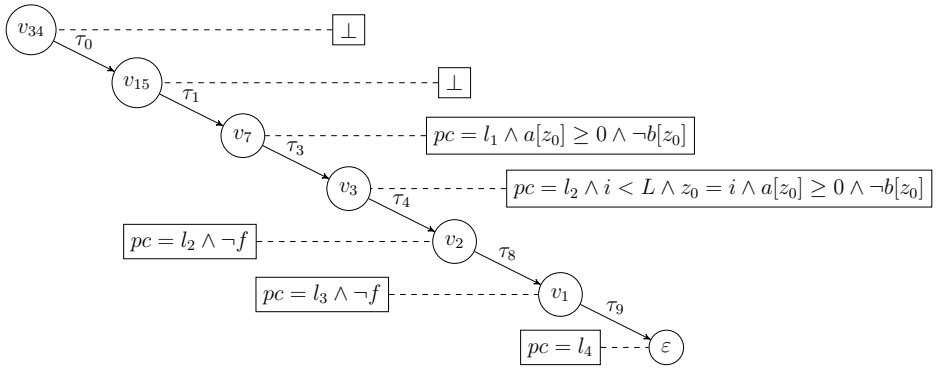


Figure 3.3. Path obtained by refining the counterexample in Figure 3.2.

3.3.4 An interpolation procedure for quantifier-free formulæ

We now describe the interpolation procedure for quantifier-free formulæ used to compute path-interpolants for refining infeasible counterexamples (as described in section 3.3.3).

First of all, recall that we assumed that quantifier-free interpolants can be computed for both $\mathcal{T}_I$ and $\mathcal{T}_E$ in section 3.1. Unfortunately, this is not sufficient to guarantee the possibility to compute quantifier-free interpolants for quantifier-free formulæ in $\mathcal{A}_I^E$. In fact, this theory can be seen as a combination of $\mathcal{T}_I$ and $\mathcal{T}_E$ with (uninterpreted) function symbols by considering arrays as function symbols and the dereference operation as function application. Negative results (such as [Brillout et al., 2010, Bruttomesso et al., 2012a]) are available in the literature showing that the addition of (uninterpreted) function symbols to theories allowing for the computation of quantifier-free interpolants prevents the existence of quantifier-free interpolants in the extended theory. Fortunately, the $\mathcal{A}_I^E$-unsatisfiable formulæ of the form $\psi_1 \wedge \psi_2$ for which an interpolant must be computed when invoking the procedure Refine are such that ψ_1 and ψ_2 satisfy certain conditions on their shape that guarantee the possibility to compute quantifier-free interpolants as stated in the following result.

Theorem 3.3.2. *Suppose that $\psi_1 \wedge \psi_2$ is an $\mathcal{A}_I^E$-unsatisfiable quantifier-free formula such that all terms of sort* `INDEX` *occurring in ψ_2 under the scope of the dereference operation $\_[\_]$ occur also in ψ_1. Then, there exists a quantifier-free formula ψ_0 such that: (i) $\psi_2 \models_{\mathcal{A}_I^E} \psi_0$; (ii) $\psi_0 \wedge \psi_1$ is $\mathcal{A}_I^E$-unsatisfiable; and (iii) all free variables occurring in ψ_0 occur both in ψ_1 and ψ_2.*

Proof. Let us call *critical* the index variables occurring both in ψ_1 and ψ_2 (by assumptions, the index variables occurring in ψ_2 under the scope of the deref-

erence operator $\_[\_]$ are critical). Without loss of generality, we may assume that ψ_1 and ψ_2 are conjunctions of dereference flat literals[4] and that for all distinct variables i, j occurring in ψ_1, we have that ψ_1 contains either the literal $i = j$ or the literal $i \neq j$. These assumptions can be justified by standard considerations. For instance, once interpolants for $\psi_1' \wedge \psi_2$ and for $\psi_1'' \wedge \psi_2$ are known, one can combine them to an interpolant for $(\psi_1' \vee \psi_1'') \wedge \psi_2$ by taking disjunction[5]. We can also assume that, whenever ψ_1 contains $i = j$, then it contains also $a[i] = a[j]$ for every array variable occurring in ψ_1; finally, if i, j are critical variables and $i = j$ is a conjunct of ψ_1, then we assume that ψ_2 contains $a[i] = a[j]$ for every array variable a occurring in ψ_2. In fact, if adding $i = j \wedge a[i] = a[j]$ to ψ_2 one gets the interpolant ψ_0, it is possible to get the interpolant back from ψ_2 by taking $i = j \rightarrow \psi_0$.

Let now ψ_1 be of the kind

$$\psi_1(\underline{i}_1, \underline{i}_0, \underline{a}_1[\underline{i}_1], \underline{a}_1[\underline{i}_0], \underline{a}_0[\underline{i}_1], \underline{a}_0[\underline{i}_0], \underline{e}_1, \underline{e}_0)$$

and ψ_2 be of the kind

$$\psi_2(\underline{i}_0, \underline{i}_2, \underline{a}_2[\underline{i}_0], \underline{a}_0[\underline{i}_0], \underline{e}_2, \underline{e}_0),$$

where $\underline{a}_1, \underline{a}_0, \underline{a}_2$ are array variables, $\underline{e}_0, \underline{e}_1, \underline{e}_2$ are element variables, and $\underline{i}_0, \underline{i}_1, \underline{i}_2$ are index variables (the $\underline{i}_0$ are the critical ones - notice that terms $\underline{a}_0[\underline{i}_2], \underline{a}_2[\underline{i}_2]$ do not occur in ψ_2). We can further separate the literals whose root predicate symbol has argument of sort `INDEX` from the literals whose root predicate has arguments of sort `ELEM`, thus ψ_1 can be rewritten as

$$\psi_1^I(\underline{i}_1, \underline{i}_0) \wedge \psi_1^E(\underline{a}_1[\underline{i}_1], \underline{a}_1[\underline{i}_0], \underline{a}_0[\underline{i}_1], \underline{a}_0[\underline{i}_0], \underline{e}_1, \underline{e}_0)$$

whereas ψ_2 as

$$\psi_2^I(\underline{i}_0, \underline{i}_2) \wedge \psi_2^E(\underline{a}_2[\underline{i}_0], \underline{a}_0[\underline{i}_0], \underline{e}_2, \underline{e}_0)$$

for ψ_g^I and ψ_g^E conjunctions of literals whose root predicate symbols have argument of sort `INDEX` and `ELEM`, respectively, and $g = 1, 2$.

Now, since a complete partition on indexes $\underline{i}_0, \underline{i}_1$ is included in ψ_1[6] and relevant index equalities have been fully propagated through array variables,

[4]Recall from the proof of Lemma 2.2.2 that a literal is dereference flat if the only terms occurring as arguments of the function symbols are variables.

[5]For a general framework covering all these transformations, the reader is pointed to [Bruttomesso et al., 2012b].

[6]In practice, this might result in a large combinatorial blow-up. Practical optimizations for the scalability of this procedure will be described in chapter 4.

it is easy to see, by using the same argument as in the proof of Lemma 2.2.2, that the inconsistency of $\psi_1 \wedge \psi_2$ implies that either

$$\psi_1^I(\underline{i}_1, \underline{i}_0) \wedge \psi_2^I(\underline{i}_0, \underline{i}_2)$$

is $\mathcal{T}_I$-unsatisfiable or

$$\psi_1^E(\underline{d}'_1, \underline{d}''_1, \underline{d}'''_1, \underline{d}_0, \underline{e}_1, \underline{e}_0) \wedge \psi_2^E(\underline{d}_2, \underline{d}_0, \underline{e}_2, \underline{e}_0)$$

is $\mathcal{T}_E$-unsatisfiable, where we used fresh element variables $\underline{d}_0, \underline{d}'_1, \underline{d}''_1, \underline{d}'''_1, \underline{d}_2$ instead of the terms $\underline{a}_0[\underline{i}_0], \underline{a}_1[\underline{i}_1], \underline{a}_1[\underline{i}_0], \underline{a}_0[\underline{i}_1], \underline{a}_2[\underline{i}_0]$, respectively. Now it is clear that we can use the available quantifier-free interpolation algorithms for $\mathcal{T}_I$ and $\mathcal{T}_E$ in order to compute the interpolant ψ_0. □

3.4 Correctness and termination

Recall that UNWIND consists of the exhaustive (non-deterministic) application of EXPAND and REFINE. We now show that UNWIND correctly establishes the safety of an array-based system when terminating.

Theorem 3.4.1. *Let* UNWIND *be applied to an array-based system* $\mathcal{S}_{\mathcal{A}_I^E}$. *If* UNWIND *reports unsafety, then* $\mathcal{S}_{\mathcal{A}_I^E}$ *is unsafe. If neither* EXPAND *nor* REFINE *can be applied to a labeled unwinding* P *of* $\mathcal{S}_{\mathcal{A}_I^E}$, *then* P *is safe and complete (and thus* $\mathcal{S}_{\mathcal{A}_I^E}$ *is safe by Theorem 3.2.1).*

Proof. The first part of the claim is obvious. For the second part, let us consider a labeled unwinding $P = (V, E, M_V, M_E)$ of $\mathcal{S}_{\mathcal{A}_I^E}$ to which neither EXPAND nor REFINE applies. We first show that P is complete. Notice that if leaves are all labeled by $\mathcal{A}_I^E$-unsatisfiable formulæ, non-leaf vertexes are a covering, and the system is complete. On the other hand, if there is a leaf labeled by an $\mathcal{A}_I^E$-satisfiable formula, one of the two sub-procedures applies unless the current labeled unwinding is recognized to be complete – according to Definition 3.2.3 in section 3.2.2 – and hence complete *tout court*. Thus, the labeled unwinding must be complete when no sub-procedure is applicable.

Finally, if P is not safe, there is a consistent vertex v whose location is l_{init}. Now, since l_{init} is not a target location, v must be a leaf; for the same reason, v is not covered by non-leaf vertexes (the location of these vertexes is not l_{init}). Thus the labeled unwinding is not complete, hence it cannot be recognized as such, and REFINE is applicable. □

This result implies the partial correctness of UNWIND. In the rest of this section, we investigate total correctness.

3.4.1 Precisely recognizing complete labeled unwindings

The first step towards the total correctness of UNWIND is to have a complete "default satisfiability procedure" for recognizing complete covers; recall Definition 3.2.3 in section 3.2.2. The default satisfiability procedure uses the "default instantiation procedure" to reduce the problem of checking the $\mathcal{A}_I^E$-satisfiability of $\exists^{A,I}\forall^I$-sentences to checking the $\mathcal{A}_I^E$-satisfiability of quantifier-free formulæ. Since a decision procedure for the latter is available (under the hypothesis that the $SMT(\mathcal{T}_I)$ and the $SMT(\mathcal{T}_E)$ problems are decidable as assumed in section 3.1), we need to find conditions under which the default instantiation procedure is complete. To formally characterize this, we need to introduce the following notion.

A class $\mathcal{C}$ of structures is *closed under substructures* if for every structure $\mathcal{M} \in \mathcal{C}$, it happens that all the sub-structures of $\mathcal{M}$ are also in $\mathcal{C}$. Any theory whose class of models is specified as the class of models of a set of universal sentences, i.e. formulæ containing no free variables obtained by prefixing a quantifier-free formula with a finite sequence of universal quantifiers, is closed under substructures by well-known results in model theory (see, e.g., [Hodges, 1993]). For example, the theory of posets (i.e. of sets endowed with a reflexive, transitive and antisymmetric relation) can be axiomatized by a set of universal sentences and it is thus closed under substructures.

Theorem 3.4.2 ([Ghilardi and Ranise, 2010a])**.** *If there are no function symbols in the signature Σ_I of $\mathcal{T}_I$ and the class $\mathcal{C}_I$ of models of $\mathcal{T}_I$ is closed under substructures, then the $\mathcal{A}_I^E$-satisfiability of $\exists^{A,I}\forall^I$-sentences is decidable.*

Proof. We claim that, under the hypotheses of the theorem, the $\mathcal{A}_I^E$-satisfiability of (3.7), i.e.,

$$\bigwedge_{\sigma\in\Sigma} \psi(\underline{i}, \underline{j}\sigma, \mathbf{a}[\underline{i}], \mathbf{a}[\underline{j}], \mathbf{c}, \mathbf{d}) \tag{3.7}$$

(where Σ denotes the set of all possible substitutions mapping the variables in $\underline{j}$ into $\underline{i} \cup \mathbf{c}$) implies the $\mathcal{A}_I^E$-satisfiability of (3.6), i.e.

$$\exists\mathbf{a}\,\exists\mathbf{c}\,\exists\mathbf{d}\,\exists\underline{i}\,\forall\underline{j}.\ \psi(\underline{i}, \underline{j}, \mathbf{a}[\underline{i}], \mathbf{a}[\underline{j}], \mathbf{c}, \mathbf{d})\,. \tag{3.6}$$

This is sufficient to show the decidability of the $\mathcal{A}_I^E$-satisfiability of $\exists^{A,I}\forall^I$-sentences since the $\mathcal{A}_I^E$-satisfiability of (3.7) is decidable by Lemma 2.2.2 and

the $\mathcal{A}_I^E$-satisfiability of (3.6) implies the $\mathcal{A}_I^E$-satisfiability of (3.7).

We consider a structure $\mathcal{M}$ which (together with an assignment to the free variables $\mathbf{c}, \mathbf{d}$) is a model of (3.7) and we derive from this a structure $\mathcal{M}'$ as follows. First, the interpretation of the sort `INDEX` in $\mathcal{M}'$ is obtained by restricting that in $\mathcal{M}$ of the same sort `INDEX` (as well as of all symbols in Σ_I) to the subset containing only the elements assigned to the variables in $\underline{i}, \mathbf{c}$. The interpretation of the symbols of Σ_E in $\mathcal{M}'$ is identical to that of $\mathcal{M}$ and the functions assigned to the $\mathbf{a}$'s in $\mathcal{M}'$ are the same of those in $\mathcal{M}$ but restricted to their domains. Since $\mathcal{C}_I$ is closed under substructures, $\mathcal{M}'$ is still an $\mathcal{A}_I^E$-model. It is easy to see that, since (3.7) is quantifier-free, the truth of (3.7) is inherited by $\mathcal{M}'$. Additionally, because of the restriction of the interpretation of the sort `INDEX`, (3.6) also holds in $\mathcal{M}'$. This concludes the proof of the claim above. □

3.4.2 Termination of Unwind

Now, that we have found conditions under which precise checks to recognize the completeness of labeled unwindings can be obtained, we focus on studying the termination of Unwind.

First of all, we notice that the termination of Unwind can be easily ensured when $\mathcal{S}_{\mathcal{A}_I^E}$ *is unsafe* by adopting suitable strategies for the application of the sub-procedure Expand. For example, a breadth-first strategy used when expanding the labeled unwinding certainly guarantees termination (the design of other strategies is mostly an implementation issue, see for instance section 4.2 or also [McMillan, 2006]).

If $\mathcal{S}_{\mathcal{A}_I^E}$ *is safe*, the termination of Unwind cannot be shown for arbitrary array-based systems since their safety problem is undecidable in general (see, e.g., [Ghilardi and Ranise, 2010a]). In the following, we investigate sufficiently restrictive conditions under which Unwind is guaranteed to terminate. In particular, we identify two sufficient conditions for this. First, a fair strategy must be used to apply Expand and Refine. Formally, a strategy is *fair* if it does not indefinitely delay the application of one of the two procedures and does not apply Refine infinitely many times to the label of the same vertex. Notice that the latter holds if there are no infinitely many non-equivalent formulæ of the form $\psi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ for a given $\underline{i}$ or, alternatively, if a refinement based on the computation of interpolants through the precise preimage is eventually applied when repeatedly refining a vertex.

The second condition for the termination of Unwind concerns the theory $\mathcal{T}_E$. To formalize this, we need to introduce some formal notions. An *exis-*

tential Σ*-sentence* is a formula containing no free variables that is obtained by prefixing a quantifier-free Σ-formula with a finite sequence of existential quantifiers. A structure $\mathcal{M}$ is *finitely generated* iff there exists a finite sub-set X of the support of $\mathcal{M}$ such that the smallest substructure of $\mathcal{M}$ containing X is $\mathcal{M}$ itself. An embedding is an injective homomorphism that preserves and reflects relations and operations. A reflexive-transitive relation $\preceq$ on a set P is a *well-quasi-order* (wqo) iff given $p_0, p_1, \ldots p_n, \ldots$ from P, there are $n < m$ such that $p_n \preceq p_m$. A *wqo-theory* [Carioni et al., 2011] is a theory $\mathcal{T} = (\Sigma, \mathcal{C})$ such that $\mathcal{C}$ is closed under substructures and finitely generated models of $\mathcal{T}$ are a well-quasi-order with respect to the relation $\preceq$ that holds between $\mathcal{M}_1$ and $\mathcal{M}_2$ whenever $\mathcal{M}_1$ embeds into $\mathcal{M}_2$. As shown in [Carioni et al., 2011], the following is a wqo-theory: it contains one sort, finitely many 0-ary and unary predicate symbols, a single binary predicate symbol $\leq$, and its class of models satisfies the following three (universal) sentences: $\forall x\,(x \leq x)$, $\forall x, y, z\,(x \leq y \wedge y \leq z \rightarrow x \leq z)$, and $\forall x, y\,(x \leq y \vee y \leq x)$, constraining $\leq$ to be interpreted as a total pre-order.

We also need the following technical result.

Lemma 3.4.1. *Let $\mathcal{T} = (\Sigma, \mathcal{C})$ be a wqo-theory and $K_0, K_1, \ldots, K_n, \ldots$ be an infinite sequence of existential Σ-sentences such that $K_n \models_{\mathcal{T}} K_{n+1}$ for all $n \geq 0$. Then, there exists $n > 0$ such that $K_n \models_{\mathcal{T}} K_{n-1}$.*

Proof. Suppose the statement does not hold. Then, for every n there exists a model $\mathcal{M}_n \in \mathcal{C}$ such that $\mathcal{M}_n \models K_n$ and $\mathcal{M}_n \not\models K_{n-1}$. Since $\mathcal{C}$ is closed under substructures and K_n is an existential sentence, we can take $\mathcal{M}_n$ to be finitely generated. Notice that truth of $\neg K_{n-1}$ is preserved by substructures because this is a universal formula (see, e.g., [Hodges, 1993]). Since $K_m \models_{\mathcal{T}} K_{n-1}$ for $m < n$, we have that $\mathcal{M}_n \not\models K_m$ for every $m < n$. Consider now the sequence $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n, \ldots$ of finitely generated models in $\mathcal{C}$. By definition of a well-quasi-order, there must be $m < n$ such that $\mathcal{M}_m$ embeds in $\mathcal{M}_n$. Then, from $\mathcal{M}_m \models K_m$ and the fact that K_m is existential, it follows that $\mathcal{M}_n \models K_m$. Contradiction. □

We are now in the position to state and prove our result on the termination of Unwind.

Theorem 3.4.3. *Let $\mathcal{S}_{\mathcal{A}_I^E}$ be an array-based system for $\mathcal{T}_I, \mathcal{T}_E$. Suppose that $\mathcal{T}_I$ satisfies the hypotheses of Theorem 3.4.2 and that the theory obtained from $\mathcal{T}_I \cup \mathcal{T}_E$ by adding the symbols in $\mathbf{v}$, seen as free function or constant symbols of appropriate sorts, is a wqo theory. Then,* Unwind *terminates when applied to $\mathcal{S}$ with a fair strategy.*

Proof. If we view the state variables $\mathbf{v} := \mathbf{a}, \mathbf{c}, \mathbf{d}$ of the array-based system $\mathcal{S}_{\mathcal{A}_I^E} = \langle \mathbf{v}; l_{\mathsf{init}}; l_{\mathsf{error}}; T \rangle$ as free (function or constants) symbols, the existential (index) closures of the formulæ (and their disjunctions) labeling the vertexes in a labeled unwinding of $\mathcal{S}_{\mathcal{A}_I^E}$ are $\exists^I$-formulæ of the form $\exists \underline{i}\, \psi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$. Thus these are existential formulæ of the wqo theory mentioned in the statement of the theorem and Lemma 3.4.1 is applicable.

If the fair strategy used to apply Expand and Refine does not terminate, it generates a sequence of labeled unwindings $P_0, P_1, P_2, \ldots$ where $P_j = (V_j, E_j, M_V^j, M_E^j)$ is such that $V_j \subseteq V_{j+1}$ and $E_j \subseteq E_{j+1}$, written as $(V_j, E_j) \subseteq (V_{j+1}, E_{j+1})$, for $j \geq 0$. In other words, we have an increasing sequence of trees of the form $(V_0, E_0), (V_1, E_1), \ldots$ Consider now the union $(V, E) = (\bigcup_k V_k, \bigcup_k E_k)$ of all the trees in the sequence. Since vertices are not refined infinitely often, we can associate with any vertex $v \in V$ its (ultimate) label $M(v)$. Let K_n be the disjunction of the labels $M(v)$ where v is a vertex of (V, E) of depth at most n: by Lemma 3.4.1, we have that $K_n \models_{\mathcal{A}_I^E} K_{n-1}$ for some $n > 0$. This means that for every vertex v in (V, E) of depth at most n, we have that $M(v) \models_{\mathcal{A}_I^E} \bigvee_{w \in C} M(w)$ where C is the set of vertexes of (V, E) of depth at most $n-1$ whose label is $\mathcal{A}_I^E$-satisfiable.

Let now i be large enough so that every non-leaf vertex of depth at most n in (V, E)—together with its ultimate label—is in P_i: we show that Unwind should have terminated after P_i has been produced. There are two cases to consider. First, C is a covering for all labeled unwinding P_j such that $P_i \subseteq P_j$ and would cause Unwind to terminate. Second, C is not a covering because C contains a leaf w. However $M(w)$ is $\mathcal{A}_I^E$-satisfiable by the definition of C and is the ultimate label of w. Now we have that $M(w) \models pc = l_{\mathsf{init}}$, otherwise our fair strategy would have added some vertices as sons of w, because locations $l \neq l_{\mathsf{init}}$ are target locations. This means that a refinement step applies to w. Since $M(w)$ is $\mathcal{A}_I^E$-satisfiable and is the ultimate label of w, this means that such refinement step must have reported the unsafety of $\mathcal{S}_{\mathcal{A}_I^E}$. □

The hypotheses of Theorem 3.4.3 are rather restrictive when it comes to the analysis of imperative programs. Fragments of arithmetic play a central role in this domain and their usage in modeling operations on array indexes prevents the applicability of Theorem 3.4.3. For an application of this result, let us consider, therefore, a different application domain, like that of broadcast protocols (see, e.g., [Delzanno et al., 1999]). These are systems composed of a finite but arbitrary number of (identical) processes that can communicate by rendez-vous (a process sends a message to another) or broadcast (a process sends a message to all the others). Any such system can be speci-

fied by an array-based system $\mathcal{S} = \langle \mathbf{v}; l_{\mathsf{init}}; l_{\mathsf{error}}; T \rangle$ for $\mathcal{T}_I$ the (pure) theory of equality (used to represent process identifiers) and $\mathcal{T}_E$ an enumerated data-type theory (representing the finite set of locations of each (identical) process) where $\mathbf{v} = \mathbf{a}, \mathbf{c}, \mathbf{d}$ and $\mathbf{a}$ contains just one function symbol (associating a process identifier to the actual location reached by the process) whereas both $\mathbf{c}$ and $\mathbf{d}$ are empty. As shown in [Ghilardi and Ranise, 2010a], it is possible to represent rendez-vous and broadcast of messages as guarded assignments in functional form (3.1). In [Carioni et al., 2011], it is shown that the theories $\mathcal{T}_I$ and $\mathcal{T}_E$ satisfy the hypotheses of Theorem 3.4.3. Thus, UNWIND behaves as a decision procedure for the safety problem of broadcast protocols. A similar result using forward reachability has been proved in [Dimitrova and Podelski, 2008]. Complexity-wise, for broadcast protocols the reachability analysis has a non-primitive recursive complexity, as stated in [Esparza et al., 1999, Delzanno et al., 1999].

It is also possible to show that UNWIND behaves as a decision procedure for the safety problem of lossy channel system systems (see, e.g., [Abdulla and Jonsson, 1996]): their representation as array-based systems can be found in [Ghilardi and Ranise, 2010a] and the fact that the latter satisfy the hypotheses of Theorem 3.4.3 is shown in [Carioni et al., 2011].

3.5 Related work

The vast majority of state-of-the-art frameworks for the formal verification of infinite-state systems is abstraction-based, and a long list of efficient techniques for the analysis of programs is available in the literature. Below, we discuss the relevant work classified according to the main technique they use as follows: predicate abstraction with counterexample guided abstraction refinement procedures, abstract interpretation, theorem proving-based, shape analysis and template-based solutions.

3.5.1 Predicate abstraction

Since the seminal paper [Graf and Saïdi, 1997], *Predicate abstraction* has become a very popular technique in software verification. One of the first approaches for software verification based on predicate abstraction and able to handle quantified predicate is in [Flanagan and Qadeer, 2002]. This solution exploits *ghost* variables, i.e., Skolem constants which are never modified by the program. Ghost variables, once the procedure terminates, are not assigned to a

precise value and hence can be universally quantified. The *index predicate* solution [Lahiri and Bryant, 2004b] fixes the number of "index variables", i.e., universally quantified variables, in order to exploit standard predicate abstraction algorithms. For such two solutions predicates are generally suggested by the user. The work in [Lahiri and Bryant, 2004a] proposes a refinement technique based on the weakest precondition, in charge of generating new intermediate annotations. The main limitation of the aforementioned approaches is their inability to generate quantified predicates. These approaches would be inefficient, therefore, on programs without quantified post-conditions or assertions like those considered in part of our experimental analysis. The generation of quantified predicates has been addressed also by Jhala and McMillan in [Jhala and McMillan, 2007], as an extension of their previous work [Jhala and McMillan, 2006]. Interpolating procedure are driven by new axioms with the goal of generating quantified predicates, called *range predicates*, representing properties for ranges of cells in the arrays. While such predicates are restricted to a particular shape, this is not the case of our technique. Invariants and predicates can also be generated by analyzing the postcondition with some patterns, like *variable aging* or *constant relaxation* [Furia and Meyer, 2010]. This approach can generate invariants for many interesting problems, like sorting algorithms. On the other hand, it cannot handle programs which require quantified invariants but do not have quantified assertions in their specifications.

Arrays can also represent a contiguous, fixed-size, portion of memory. For this class of programs, blasting every cell of the array as a single, uncorrelated variable results in inefficient procedures, as pointed out by in [Armando et al., 2007b, Armando et al., 2007a], which present an abstraction-refinement procedure for linear programs with fixed-size arrays.

3.5.2 Abstract interpretation

The approach described in this chapter aims at developing a sound analysis procedure at the price of non-termination. Our solution does not suffer from the loss of precision deriving from the use of approximation techniques and, upon termination, returns either an invariant, which is both safe and inductive, or a real counterexample. *Abstract Interpretation* (AI) approaches target efficiency, i.e., they aim to generate inductive (but not necessarily safe) facts at compile-time. The application of widening operators, required to ensure the convergence of the analysis, may cause loss of precision, though, with the result that inferred inductive properties might be too weak to prove the absence of paths violating a given property.

AI solutions rely on the availability of some *abstract domains* for inferring invariants. An abstract domain can be thought of as a (fragment of a) theory [Gulwani and Tiwari, 2006] identifying a class of formulæ over which the concrete semantics of the input program is abstracted. Since the seminal paper [Cousot and Cousot, 1977], several domains (such as interval arithmetic [Cousot and Cousot, 1977], octagons [Miné, 2006], octahedra [Clarisó and Cortadella, 2007], and convex-polyhedra [Cousot and Halbwachs, 1978]) have been studied in order to reason about different properties of programs.

AI analysis for arrays can be performed by associating one abstract value to each cell of the array or by *smashing* array variables, i.e., using one abstract value representing all the possible values of the array [Blanchet et al., 2002]. The first approach is precise but extremely inefficient while the second, on the contrary, is much more efficient at the price of (greatly) degrading precision. Other approaches segment either syntactically [Gopan et al., 2005, Halbwachs and Péron, 2008] or semantically [Cousot et al., 2011] an array and assign to each segment an abstract value.

The long-term project CODE CONTRACTS[7] carried on at Microsoft Research has obtained very good results and its value in both the academic and industrial scenarios should not be neglected. The project supports static verification of programs with several analysis tools, many of which are based on AI techniques such as CLOUSOT.

It is worth to notice that abstract interpretation and CEGAR-based approaches are not mutually exclusive. They have been successfully combined, for example, in [Albarghouthi et al., 2012a]. Our BOOSTER framework, discussed in chapter 8, combines as well abstract interpretation with (our) LAWI solution.

3.5.3 Theorem Proving

Inference of quantified array properties is the goal of the techniques in [McMillan, 2008, Kovács and Voronkov, 2009, Hoder et al., 2010]. The generation of quantified predicates relies on the use of saturation-based theorem proving (i.e. resolution extended with inferences to reason about equalities) combined with interpolation [McMillan, 2008, Hoder et al., 2010] or the solution of recurrence relations [Kovács and Voronkov, 2009].

Invariants produced by these approaches may be more expressive than those found by our technique; for instance, they may contain alternations of quan-

[7] `http://research.microsoft.com/projects/contracts`

tifiers. Indeed, considering a larger class of properties makes the problem of avoiding divergence even more acute than in our setting. The situation is further complicated by the fact that saturation-based theorem provers need to be instructed with axioms for handling arithmetic and this may, in practice, further contribute to the non-termination of the inference process (theoretically, satisfiability of arbitrary first-order formulæ is semi-decidable). Instead, our approach relies on SMT-Solvers to take care of the arithmetic operations arising from the analysis of programs. This, combined with the heuristic of Term Abstraction (see section 4.1.1), greatly helps to avoid divergence in practice as shown by the experiments in section 4.2.

3.5.4 Shape analysis and Separation Logic

Heap manipulating programs are the target of *shape analysis* and *separation logic* approaches. Their goal is to infer a conservative characterization of the structure of the heap at each point of the program (see, e.g., [Reynolds, 2002, Hind, 2001]). Objects allocated on the heap are represented by a *heap graph*, where vertices are object allocated on the heap and edges are pointers accessing the objects [Chase et al., 1990]. Abstraction of these graphs can be done by using a three-value logic [Sagiv et al., 1999] or extending predicate abstraction to work with heap predicates [Podelski and Wies, 2005].

While the goal of these techniques is to provide efficient and, at the same time, expressive analysis for pointers and unbounded data structures, our goal is to discover invariants for unbounded array elements.

3.5.5 Template-based approaches

Template based approaches (e.g., [Beyer et al., 2007b,Srivastava and Gulwani, 2009] to cite a few) may infer properties which are more expressive than the properties inferred by SAFARI, but are limited to those matching a given pattern. On the contrary our solution does not require in general user intervention in specifying templates for invariant: the only interaction of the user with the tool is by suggesting an appropriate term abstraction list whenever the tool seems to diverge. Recently, [Larraz et al., 2013] presents a constraint-based invariant generation technique suited for the synthesis of quantified array invariants. This approach is SMT-based and uses non-linear constraints. It can synthesize invariants containing just one quantified variable and does not apply to nested loops. Our approach, instead, is not limited to invariants containing one quantified variables and can be applied to programs with nested loops, as

witnessed by the experiments in section 4.2.

3.6 Summary

In this chapter we presented an extension of the Lazy Abstraction with Interpolants framework [McMillan, 2006] suitable for the analysis of programs handling arrays.

Our technique is based on a backward reachability procedure for array-based transition systems [Ghilardi and Ranise, 2010a] interleaved with a CEGAR procedure. Distinguishing features of our technique are the generation of quantified predicates by a refinement phase using quantifier-free interpolants.

In our approach, the transition relation is preliminary flattened. This process ensures that the arrays handled by the program will be only indexed by existentially quantified variables (section 3.1). State-space is explored in a backward fashion, according to the Model-Checking Modulo Theories framework described in [Ghilardi and Ranise, 2010a]. In our case, the backward reachability analysis is enriched with abstraction and refinement procedures combined following the CEGAR paradigm (section 3.2). In particular, the refinement procedure includes a quantifier-instantiation step turning quantified counterexamples in quantifier-free formulæ over arrays. We showed that the fragment including our counterexamples admits quantifier-free interpolation. This allows to depict an incremental refinement procedure generating the desired safe inductive invariants (section 3.3). In addition, thanks to the preprocessing step, the safe inductive invariant will contain existentially quantified variables. Recalling our backward exploration strategy, the safe inductive invariant we obtain once our new framework terminates is the negation of a *universally quantified* safe inductive invariant proving the safety of the input program. We also discussed some hypothesis ensuring the termination of our new backward, CEGAR-based, reachability analysis (section 3.4).

Next chapter will discuss engineering strategies for an effective implementation of the framework presented in this thesis.

3.6.1 Related publications

The results reported in this chapter have been published in the following papers:

- F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy abstraction with interpolants for arrays. In N. Bjørner and A. Voronkov,

editors, *LPAR*, volume 7180 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2012.

- F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. An extension of lazy abstraction with interpolation for programs with arrays. *Formal Methods in System Design*, 45(1):63–109, 2014.

Chapter 4

SAFARI – SMT-based Abstraction For Arrays with Refinement by Interpolation

This chapter describes an efficient implementation of the framework presented in chapter 3 in a model checker called SAFARI – "SMT-Based Abstraction For Arrays with Interpolants". The LAWI framework presented in chapter 3 has several critical points requiring suitable heuristics for achieving practical effectiveness. The two most important are the following:

- Quantifier handling. Our LAWI framework works with quantified formulæ. Safety tests are checked by evaluating the safety of $\exists^I$-formulæ which satisfiability is decidable and for which SMT-solvers offer efficient decision procedures. For fix-point tests the situation is complicated since these are satisfiability problems of $\exists^{A,I}\forall^I$-formulæ over the theory of arrays, generally falling outside known decidable class (e.g., [Ge and de Moura, 2009, Bradley et al., 2006]).

- Interpolants are not unique. Several variables determines how an abstract system will be refined: internal heuristics of the solver used to detect the unsatisfiability of a given formula representing an infeasible counterexample and the procedure (or, more precisely, the labeling strategy) adopted to compute the interpolants and the most relevant ones. These parameters govern the nature of the outcoming interpolants (strong, weak, etc.). Computing good interpolants is vital for developing effective model-checkers, and generally one cannot rely on the interpolants computed from an interpolation theorem prover (or an SMT-solver). This because

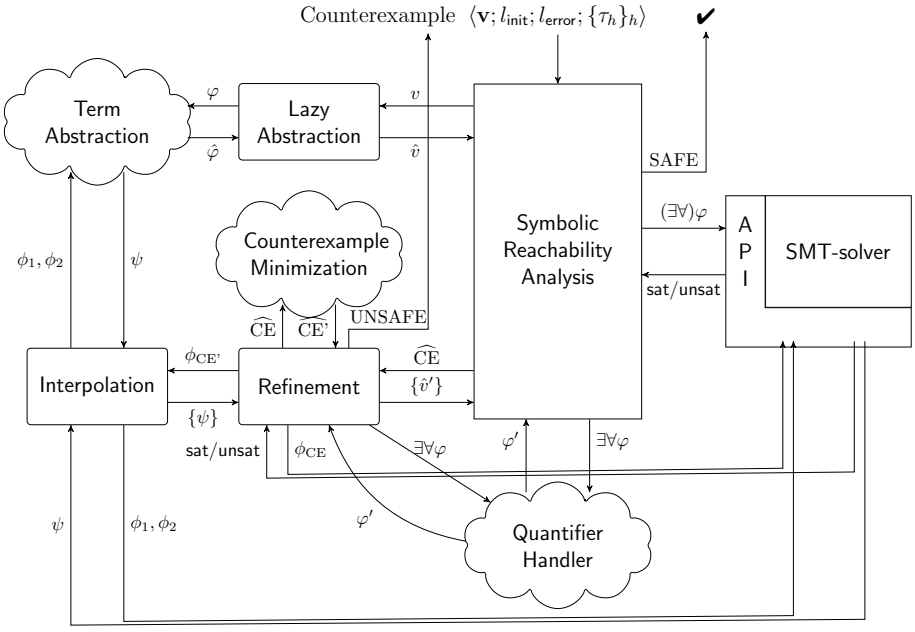


Figure 4.1. The architecture of SAFARI.

the interpolation prover, which is not aware of the nature of the program, cannot select blindly the best setting for producing "good" interpolants.

These two problems require practical engineering choices in order to develop an effective model checker. In this chapter we present the architecture of SAFARI, discuss the heuristics implemented to make it scaling on non-trivial state-of-the-art benchmarks and analyze its thorough experimental evaluation.

4.1 Implementation and heuristics

Given the tight link with the content of chapter 3, we assume the same background notions introduced in section 3.1.

The architecture of the tool is depicted in Figure 4.1. Modules drawn as square boxes represent usual modules of CEGAR-based model checkers with interpolation-based refinement. Those drawn as clouds constitutes the novel features of our tool.

Our tool maintains and modifies a labeled unwinding (V, E, M_V, M_E) (see section 3.2.1 for a formal definition). We assume a total ordering $\preceq \subseteq V \times V$ respecting the ancestor relation. In our implementation, each vertex $v \in V$ is

flagged as *free*, *covered* or *locked*. When created, all the vertices are free. A vertex v can become covered only if i) there exists a set of free vertices C such that (3.4) holds, i.e.

$$M_V(v)^{\exists} \models_{\mathcal{A}_I^E} \bigvee_{w \in C} M_V(w)^{\exists}$$

where $w \preceq v$ for all $w \in C$, and ii) all the vertices from v to ε are free. A vertex becomes locked when one of its ancestors gets covered.

The Symbolic Reachability Analysis module implements two procedures: Expand and Reduce. The Expand procedure is in charge to expand the labeled unwinding, as explained in section 3.3. The practical implementation of this procedure, however, deviates from the high-level description provided in the previous sections by introducing some important optimizations. In our implementation Expand is applied only to free leaves. Every new leaf w generated by a vertex v is labeled with the preimage of $M_V(v)$ along the transition whose matrix is associated to $M_E(w, v)$. This allows to discover immediately trivial infeasible paths, i.e., those for which the preimage is $\mathcal{A}_I^E$-unsatisfiable. The choice of the leaf to expand is also subject to several optimizations. As will be detailed later, the efficiency of the tool greatly depends on its ability to perform covering tests. Such tests are based on instantiation procedures whose complexity might badly affect the overall performance of safari. Also the exploration strategy (i.e. the selection of the leaves to expand) strongly affects the performance of the tool. We will describe the exploration strategy implemented in safari later in section 4.1.3, when heuristics and optimizations for efficient covering checks will be discussed. The other procedure implemented by this module, namely Reduce, is in charge to limit the growth of the labeled unwinding. It works by checking the vertices of the labeled unwinding with the goal to find the covered or locked ones. Reduce is eagerly applied before and after the Expand procedure. When applied before the expansion of the labeled unwinding, Reduce checks if any vertex on the path from ε to the leaf selected for expansion is covered, starting from ε. Its application after the generation of the new leaves avoid their processing in case they are already covered. Indeed, only free newly generated vertices are passed to the Lazy Abstraction module. Given an abstracted leaf $\hat{v}$, it is checked if $M_V(\hat{v}) \wedge pc = l_{\mathsf{init}}$ is $\mathcal{A}_I^E$-satisfiable. If so, the path from ε to $\hat{v}$, represented as $\widehat{\mathrm{CE}}$ in Figure 4.1, is passed to the Refinement module. If all the leaves are flagged as covered or locked, the labeled unwinding is complete (recall Definition 3.2.3) and the set of free vertices is the covering associated to it. In this case, safari reports that the system is *safe*.

The Lazy Abstraction module is in charge to abstract labels of vertices in the unwinding. Remember that for every vertex v, $M_V(v)$ is a quantifier-free formula of the kind $\psi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ such that $M_V(v) \models_{\mathcal{A}_I^E} pc = l$ for some location l. This module returns a vertex $\hat{v}$ such that $M_V(\hat{v}) \models_{\mathcal{A}_I^E} pc = l$ and $M_V(v) \models_{\mathcal{A}_I^E} M_V(\hat{v})$.

The Refinement module implements the procedure described in section 3.3.2. It takes as input a sequence of transitions representing a candidate counterexample, and it is in charge to generate a formula attesting its feasibility. If this module fails (i.e. the formula is unsatisfiable), then the Interpolation module comes into play, as in standard interpolation-based refinement procedures. In case the (external) SMT-solver implements interpolation procedures, the Interpolation module can be bypassed by asking interpolants to the external tool. An abstract interface provides an API to separate the actual SMT-solver used and the services which are requested by SAFARI. The interface with external tools is based on the SMT-LIB v.2 standard [Ranise and Tinelli, 2006]. Refining a path might result in uncovering some vertices. Refining a vertex in the covering set C triggers a procedure that checks if the covering relation (3.4) still holds or not, and modifies the labeled unwinding as a consequence of this fact: if a vertex v was covered by a refined vertex w, and this covering relation does not hold anymore, v is considered again as a free vertex, with any locked descendant.

4.1.1 Term Abstraction

State-of-the-art interpolation procedures seldom allow the convergence of the model-checker on tricky examples. Divergence due to the inability of interpolation algorithms to come up with the "right" predicate has been already discussed in [Jhala and McMillan, 2006, Jhala and McMillan, 2007] in the context of verification of programs with scalar variables. Here, we propose a technique, called *Term Abstraction*, to tune interpolation algorithms in presence of array variables. The heuristic is implemented by the module Term Abstraction in the architecture of Figure 4.1 and its goal is to compute (whenever possible) an interpolant where a certain set T of terms (called *undesired terms*), which are responsible for keeping interpolants too specific for the analyzed counterexample, do not occur. Ultimately, abstracting away undesired terms in T aims to avoid the divergence of the sequence of interpolants generated during unwinding calls. In particular, Term Abstraction is based on the preprocessing technique described in section 2.3.1 that rewrite formulæ of the form $\psi(\cdots \mathbf{a}[\mathbf{c}] \cdots)$ to $\exists \underline{j}(\underline{j} = \mathbf{c} \wedge \psi(\cdots \mathbf{a}[\underline{j}] \cdots))$. More precisely, term abstraction works as follows.

Suppose we are given an unsatisfiable formula $\psi_1 \wedge \psi_2$ and the set $T = \{t_1, \ldots, t_n\}$ of undesired terms. We iteratively check if $\psi_1(c_i/t_i) \wedge \psi_2(d_i/t_i)$ is unsatisfiable, for c_i and d_i being fresh constants. If this is the case, we substitute ψ_j with $\psi_j(c_i/t_i)$ for $j = 1, 2$. Eventually, we are left with an unsatisfiable formula $\psi_1 \wedge \psi_2$, where some of the undesired terms in T might have been removed: the interpolant of ψ_1 and ψ_2, which can be computed with available interpolation procedures, is also likely not to contain the eliminated terms. SAFARI is capable to automatically compute a set of undesired terms to from the input transition system by identifying loop iterators, variables representing the lengths of the arrays, or loop bounds. Alternatively, the user can suggest terms to be put in the set of undesired terms. The experimental evaluation of SAFARI in section 4.2 shows that Term Abstraction plays a crucial role in the success of SAFARI.

Example 4.1.1. Consider location l_2 in Figure 2.2 corresponding to the end of the first loop in the Running procedure of Figure 2.1. SAFARI has to generate the following invariant:

$$pc = l_2 \;\rightarrow\; \forall z_0.\,((\; 0 \leq z_0 \wedge z_0 < L \;) \rightarrow (\; a[z_0] \geq 0 \leftrightarrow b[z_0] \;)) \;. \tag{4.1}$$

Key to generate this invariant is Term Abstraction. In the following, we explain how this is done. Consider the counterexample represented by the sequence of transitions $\tau_0, \tau_3, \tau_4, \tau_8, \tau_9$, generated by SAFARI during the verification of the Running procedure. To generate (4.1), we can consider the following two partitions:

$$B := \begin{pmatrix} mov(l_0, l_1, 1) \wedge i^{(1)} = 0 \wedge id(f, a[z_0], b[z_0], 1) \wedge \\ mov(l_1, l_2, 2) \wedge i^{(2)} = 0 \wedge i^{(1)} \geq L \wedge f^{(2)} \wedge id(a[z_0], b[z_0], 2) \wedge \end{pmatrix}$$

$$A := \begin{pmatrix} mov(l_2, l_2, 3) \wedge a^{(2)}[z_0] \geq 0 \wedge \neg b^{(2)}[z_0] \wedge i^{(2)} < L \wedge \\ z_0 = i^{(2)} \wedge i^{(3)} = i^{(2)} + 1 \wedge \neg f^{(3)} \wedge \\ mov(l_2, l_3, 4) \wedge i^{(3)} \geq L \wedge id(i, f, 4) \wedge \\ mov(l_3, l_4, 5) \wedge \neg f^{(4)} \wedge id(i, f, 5) \end{pmatrix}$$

An interpolant for these partitions is $I_1 := i^{(2)} < L$ since $A \models_{\mathcal{A}_I^E} I_1$ and $I_1 \wedge B$ is $\mathcal{A}_I^E$-unsatisfiable. Unfortunately, I_1 cannot be generalized to a quantified invariant as it contains no index variable.

Now, let $T = \{L, i\}$ be the set of undesired terms. The term abstraction procedure checks the unsatisfiability of $A(c/L) \wedge B(d/L)$ for the fresh constants

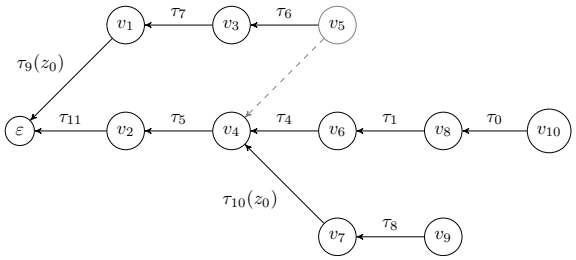


Figure 4.2. Part of the labeled unwinding for the Running procedure. $M_V(v_{68}) \wedge pc = l_I$ is $\mathcal{A}_I^E$-satisfiable and $M_V(v_{35})^{\exists} \models_{\mathcal{A}_I^E} M_V(v_{31})^{\exists}$.

c and d. The resulting formula is satisfiable, the procedure restores the original formulæ A and B, and checks whether $A(c/i^{(2)}) \wedge B(d/i^{(2)})$ is unsatisfiable. In this case it succeeds and it is thus able to generalize over the variable i. The interpolant produced in this case is $I_2 := z_0 < L$. Beside being a correct interpolant for the two original partitions, since $A \models_{\mathcal{A}_I^E} I_2$ and $I_2 \wedge B$ is $\mathcal{A}_I^E$-unsatisfiable, I_2 can be generalized to a quantified property that constitutes one of the building blocks of (4.1).

4.1.2 Minimizing counterexamples

It is useful for Refinement to apply a *minimization* procedure to counterexamples with the goal to compute interpolants from a minimal (unsatisfiable) suffix of a trace containing the atom $pc^{(n)} = l_I$. We illustrate the advantages of this by considering the following situation. Consider (part of) the labeled unwinding depicted in Figure 4.2, generated by SAFARI while analyzing the Running procedure in Figure 2.1. $M_V(v_{68}) \wedge pc = l_I$ is $\mathcal{A}_I^E$-satisfiable, and v_{31} covers v_{35} since

$$M_V(v_{31}) := pc = l_1 \wedge i < L \wedge z_0 \neq z_1 \wedge a[z_1] \geq 0 \wedge z_1 = i$$
$$M_V(v_{35}) := \begin{pmatrix} pc = l_1 \wedge i < L \wedge z_0 \neq z_1 \wedge a[z_1] = 0 \wedge z_1 = i \wedge \\ b[z_0] \wedge z_0 = 0 \wedge L > 0 \wedge L \leq i + 1 \end{pmatrix}$$

The counterexample is represented by the following formula:

$$\begin{aligned} &mov(l_0, l_1, 1) \wedge i^{(1)} = 0 \wedge id(f, a[z_0], a[z_1], b[z_0], b[z_1], 1) \wedge \\ &mov(l_1, l_1, 2) \wedge z_0 \neq z_1 \wedge i^{(2)} = i^{(1)} + 1 \wedge i^{(1)} > L \wedge \\ &\quad z_1 = i^{(1)} \wedge a^{(1)}[z_1] \geq 0 \wedge id(f, a[z_0], b[z_0], 2) \wedge \end{aligned}$$

$$
\begin{aligned}
&mov(l_1, l_2, 3) \wedge i^{(3)} = 0 \wedge L \leq i^{(2)} \wedge f^{(3)} \wedge id(a[z_0], a[z_1], 3) \wedge \\
&mov(l_2, l_2, 4) \wedge a^{(3)}[z_0] \geq 0 \wedge \neg b^{(3)}[z_0] \wedge i^{(3)} < L \wedge \\
&\quad z_0 = i^{(3)} \wedge i^{(4)} = i^{(3)} + 1 \wedge \neg f^{(4)} \wedge \\
&mov(l_2, l_3, 5) \wedge L \leq i^{(4)} \wedge id(i, f, 5) \wedge \\
&mov(l_3, l_4, 6) \wedge \neg f^{(5)} \wedge id(i, f, 6)
\end{aligned}
$$

The analysis of this counterexample can produce two different sets of interpolants:

$$
\begin{aligned}
&\quad \{\bot\} \\
&mov(l_0, l_1, 1) \wedge i^{(1)} = 0 \wedge id(f, a[z_0], a[z_1], b[z_0], b[z_1], 1) \wedge \\
&\quad \{i^{(1)} > z_0\} \\
&mov(l_1, l_1, 2) \wedge z_0 \neq z_1 \wedge i^{(2)} = i^{(1)} + 1 \wedge i^{(1)} > L \wedge \\
&\quad z_1 = i^{(1)} \wedge a^{(1)}[z_1] \geq 0 \wedge id(f, a[z_0], b[z_0], 2) \wedge \\
&\quad \{z_0 < i^{(2)} \wedge z_0 \geq 0\} \\
&mov(l_1, l_2, 3) \wedge i^{(3)} = 0 \wedge L \leq i^{(2)} \wedge f^{(3)} \wedge id(a[z_0], a[z_1], 3) \wedge \\
&\quad \{z_0 < L \wedge i^{(3)} \leq z_0\} \\
&mov(l_2, l_2, 4) \wedge a^{(3)}[z_0] \geq 0 \wedge \neg b^{(3)}[z_0] \wedge i^{(3)} < L \wedge \\
&\quad z_0 = i^{(3)} \wedge i^{(4)} = i^{(3)} + 1 \wedge \neg f^{(4)} \wedge \\
&\quad \{\top\} \\
&mov(l_2, l_3, 5) \wedge L \leq i^{(4)} \wedge id(i, f, 5) \wedge \\
&\quad \{\top\} \\
&mov(l_3, l_4, 6) \wedge \neg f^{(5)} \wedge id(i, f, 6) \\
&\quad \{\top\}
\end{aligned}
$$

or

$$
\begin{aligned}
&\quad \{\bot\} \\
&mov(l_0, l_1, 1) \wedge i^{(1)} = 0 \wedge id(f, a[z_0], a[z_1], b[z_0], b[z_1], 1) \wedge \\
&\quad \{\bot\} \\
&mov(l_1, l_1, 2) \wedge z_0 \neq z_1 \wedge i^{(2)} = i^{(1)} + 1 \wedge i^{(1)} > L \wedge \\
&\quad z_1 = i^{(1)} \wedge a^{(1)}[z_1] \geq 0 \wedge id(f, a[z_0], b[z_0], 2) \wedge
\end{aligned}
$$

$$
\begin{aligned}
&\{z_0 < i^{(2)} \wedge L \leq z_0 + 1\} \\
&mov(l_1, l_2, 3) \wedge i^{(3)} = 0 \wedge L \leq i^{(2)} \wedge f^{(3)} \wedge id(a[z_0], a[z_1], 3) \wedge \\
&\{z_0 = L - 1\} \\
&mov(l_2, l_2, 4) \wedge a^{(3)}[z_0] \geq 0 \wedge \neg b^{(3)}[z_0] \wedge i^{(3)} < L \wedge \\
&z_0 = i^{(3)} \wedge i^{(4)} = i^{(3)} + 1 \wedge \neg f^{(4)} \wedge \\
&\{L \leq i^{(4)}\} \\
&mov(l_2, l_3, 5) \wedge L \leq i^{(4)} \wedge id(i, f, 5) \wedge \\
&\{\top\} \\
&mov(l_3, l_4, 6) \wedge \neg f^{(5)} \wedge id(i, f, 6) \\
&\{\top\}
\end{aligned}
$$

The analysis of the first counterexample allows for the refinement of vertices v_4, v_9, and v_{31}. The analysis of the second counterexample permits the deletion of vertex v_{31}, as the new label is unsatisfiable, and the refinement of vertices v_9, v_4, and v_2. Notice that the second case has the drawback of "uncovering" vertex v_{35}, that, before the refinement, was covered by v_{31} since

$$M_V(v_{35})^{\exists} \models_{\mathcal{A}_I^E} M_V(v_{31})^{\exists}.$$

After the refinement such a relation does not hold anymore and v_{31} can be explored again.

Minimizing the counterexample aims at saving and preserving as much as possible the labeled unwinding. In fact, in the situation considered above, while the first set of interpolants refines only a small portion of the labeled unwinding, the second one modifies a substantial part of the unwinding and destroys part of it. The flip side of this heuristic is that postponed inconsistencies in the data-flow might appear again in counterexamples generated by later calls of Unwind, constituting the only unsat core of infeasible formula from which interpolants will be computed. In this case, the new set of interpolants would refine (and maybe destroy) the already specialized and well-refined peripheral parts of the labeled unwinding. In practice, our experience suggests that minimizing counterexamples pays off in most situations.

4.1.3 Instantiating universal quantifiers

The presence of quantified formulæ can be problematic and requires particular attention in several phases of the analysis. Quantified formulæ arise while checking covering tests and the feasibility of counterexamples. In particular, given the eager application of the REDUCE procedure, the vast majority of SAFARI execution time is spent for checking covering relations. As stated in section 3.2.2, a vertex v is covered by a set of vertices C iff

$$M_V(v)^{\exists} \models_{\mathcal{A}_I^E} \bigvee_{w \in C} M_V(w)^{\exists} \tag{4.2}$$

holds or, dually, if

$$M_V(v)^{\exists} \wedge \bigwedge_{w \in C} \neg \left(M_V(w)^{\exists} \right) \tag{4.3}$$

is $\mathcal{A}_I^E$-unsatisfiable. Stack-handling procedures available in state-of-the-art SMT-Solvers allows to perform such a test in an incremental way, asserting few formulæ representing the labels of the vertices in the set C at a time. As discussed in section 3.2.2, (4.3) is a formula of the form

$$\exists \mathbf{a}\, \exists \mathbf{c}\, \exists \mathbf{d}\, \exists \underline{i}\, \forall \underline{j}.\ \psi(\underline{i}, \underline{j}, \mathbf{a}[\underline{i}], \mathbf{a}[\underline{j}], \mathbf{c}, \mathbf{d})\,, \tag{4.4}$$

where the $\underline{i}$ are the `INDEX` variables of the vertex v and $\underline{j}$ comes from the `INDEX` variables of vertices w. As said in section 3.2.2, SAFARI deals with formulæ of the form (4.4) by using an (incomplete) satisfiability procedure based on the instantiation of $\underline{j}$ over the set $\underline{i} \cup \mathbf{c}$ of variables. Considering all possible instances becomes soon infeasible as they are $|\underline{j}|^{|\underline{i} \cup \mathbf{c}|}$. Several heuristics are integrated in SAFARI to efficiently handle this instantiation process, part of which are inherited from the tool MCMT [Ghilardi and Ranise, 2010b, Ghilardi et al., 2009]. We discuss them in the rest of this section.

4.1.4 Exploration strategy

This heuristic addresses the problem of limiting the growth of the length of the tuple $\underline{j}$ of variables; recall that $\underline{j}$ represents, intuitively, the `INDEX` variables of the labels $M_V(w)$ in (4.3).

With standard exploration strategies, such as breadth- or depth-first search, the number of index variables labeling the leaves might grow very quickly. Notice that it is possible to predict the number e_k of (implicitly existentially quantified) index variables occurring in the formulæ labeling the vertex v_k in

a path of the form $\pi = v_0 \to \cdots \to v_m$ with $v_m = \varepsilon$ by simply counting the existentially quantified index variables in $\tau_{k+1} \wedge \cdots \wedge \tau_m$ from (3.8). In fact, the number of index variables that will occur in the formula labeling v_k after the update (3.18) is bounded by e_k, because it is derived from the interpolants computed along the path π above.

Heuristics [Ghilardi and Ranise, 2009, Ghilardi and Ranise, 2010b] designed to reduce the number of index variables in preimages developed for the backward reachability procedure of MCMT can also be put to productive use in SAFARI. These heuristics affect the selection of leaves in the EXPAND procedure, promoting the expansion of leafs with a small number of index variables. SAFARI keeps an ordered list of leaves of the tree. The ordering of the leaves is firstly based on the number of `INDEX` variables, and secondly, if the number of `INDEX` variables is equal, on the $\preceq$ relation introduced in previous section. The effect of maintaining such a list is that EXPAND works always on a leaf with the smallest number of variables. Such a smart exploration strategy helps also during refinement, where quantified queries (expressing trace feasibility) are Skolemized and instantiated, thus producing equisatisfiable quantifier-free queries on which interpolation algorithms are executed.

4.1.5 Filtering instances

Adopting a smart exploration strategy helps in alleviating the burdens on the default quantifier instantiation procedure described in section 3.2.2. Even if the problem of checking satisfiability of quantified formulæ attracted a lot of interest recently (e.g., [Ge and de Moura, 2009, Ge et al., 2009, de Moura and Bjørner, 2007]), efficient solutions have been implemented only in few SMT-Solvers. We describe here another optimization devised for reducing the impact of our default instantiation procedure on the performances of SAFARI even more. This other optimization plays a significant role in the instantiation process, especially when checking covering of vertices, aims to reducing the instantiations performed for each covering test. Such optimization is based on the *filtering modulo enumerated data-type* [Ghilardi and Ranise, 2009] heuristics. They cut the number of instantiations of the universally quantified variables by exploiting cheap checks involving information cached in specific data-structures used to represent formulæ.

4.1.6 Primitive differentiated form

SAFARI inherits from MCMT the feature to keep all formulæ labeling vertices of the unwinding in a primitive differentiated form. An $\exists^I$-formula $\exists \underline{i}.\phi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ is *primitive* iff it is a conjunction of literals and is *differentiated* iff it contains the negative literal $i_k \neq i_l$ for every $i_k, i_l \in \underline{i}$. Notably, this format avoids the computationally expensive enumeration of partitions in the interpolation algorithm described in section 3.3. Primitive differentiated form helps also in reducing the number of possible instantiations while checking the unsatisfiability of formulæ of the form (4.4).

4.2 Experiments

We now present an experimental evaluation of SAFARI. We have run SAFARI against safety problems that require reasoning on arrays of unknown length (the benchmarks are illustrated in section 4.2.1). The goal of the experimental analysis is to measure the impact of the heuristics *Term Abstraction* (TA) and *Counterexample Minimization* (CM) discussed in section 4.1.1 and section 4.1.2, respectively (our findings are reported in section 4.2.2), showing that they play a central role in making SAFARI effective on non-trivial procedures.

4.2.1 Benchmarks

Our problems are divided in two benchmark suites:

- SUITE 1 consists of 13 of the 28 problems (both safe and unsafe) considered in [Dillig et al., 2010]. The programs in the problems perform simple manipulations on arrays; e.g., copying an array into another, concatenating two arrays, and swapping the content of two arrays. The safety properties are expressed by loops containing quantifier-free assertions (similarly to what is done in Figure 2.1 for the procedure Running). Each problem in SUITE 1 is labeled by "Dn" where n is a natural number used to identify the problem in [Dillig et al., 2010]. Since our tool is capable of natively supporting quantified assertions (such as (2.8) for the procedure Running), from each problem "Dn" we have derived a new (equivalent) problem identified with "QDn" by replacing the loop (or loops) encoding the safety property with the corresponding quantified

property. There are no problems "QD06" and "QD17" since the quantified properties require the use of divisibility predicates in linear arithmetic or the introduction of an alternation of quantifiers. Both cases are beyond the expressiveness of the language currently taken in input by SAFARI.

There are two reasons for the exclusion of 15 problems in [Dillig et al., 2010]. First, some of the problems in [Dillig et al., 2010] require interpolants over $\mathcal{LIA}$, i.e., linear arithmetic over the integers. $\mathcal{LIA}$ is not supported by OpenSMT [Bruttomesso et al., 2010]. Second, the remaining problems have been discarded because of the presence of C functions, such as `buffer_size`, that are not related to the kind of (quantified) array properties of interest to us in this work.

- SUITE 2 contains 25 programs taken from several sources, e.g., the benchmark suite of BOOGIE[1] and WHY3,[2] papers [Armando et al., 2007b,Hoder et al., 2010] on tools related to SAFARI, books on algorithms and data structures (such as [Wirth, 1978]), standard C string functions library, and problems suggested by experts in the area. Each program generates both a safe and an unsafe problem; the latter obtained from the former by manually inserting a bug in the problem. The programs in SUITE 2 can be briefly described as follows:
  - *binarySort* is an implementation of the "binary sort" algorithm in [Wirth, 1978]. We check that, once the procedure terminates, the array is sorted.
  - *bubbleSort* is an implementation of the "bubble sort" algorithm in [Armando et al., 2007b]. We check that, once the procedure terminates, the array is sorted.
  - *comp* implements the `strcmp` function in [Hoder et al., 2010] for comparing the content of two arrays. This function returns `true` if the two input arrays are equal. We check that if the procedure returns `true`, the two input arrays are indeed equal.
  - *compM* is a modified version of *comp* where the first equal segment of two arrays is copied in a third one. This function returns `true` if the two input arrays are equal. We check that if the procedure returns `true`, the two input arrays are indeed equals and also that the local copy of the array is equal to the input array.

[1] `http://research.microsoft.com/en-us/projects/boogie/`
[2] `http://proval.lri.fr/`

- *copy* implements the `strcpy` function in [Hoder et al., 2010] for copying the content of an array into another. The property we check is that, at the end of the procedure, the input array has been correctly copied in the returned one.
- *copyN* is a modified version of *copy* where the content of the input array is copied in N arrays (one at a time) before being copied in the last array. We check that, in the end, the N-th copied array is equal to the first one.
- *find* implements the linear search algorithm in [Hoder et al., 2010]. Such function returns the smallest index of the array where the element of interest is stored. We check that if the procedure returns a value bigger than the size of the array, the array does not contain the given element to search for.
- *findTest* is an extended version of *find* with an extra loop that checks if the returned index is the smallest one storing the given element that has been searched for. If so the function returns true. We check that the function always returns such a value.
- *heapArr* - Benchmark where the heap (abstracted as an array) is modified only in some parts. Since the postcondition asserts facts on a bigger portion, the tool has to infer that for any position outside the modified ones, the heap remained untouched. (This example has been kindly suggested by K. Rustan M. Leino).
- *init* implements the procedure in [Hoder et al., 2010] to initialize all the cells of an array to some value. We check that, at the end of the procedure, the array has been correctly initialized.
- *initTest* is an extended version of *init* with an extra loop checking that the array has been initialized. This function returns true if the extra loop does not find any error. We verify that the procedure always returns true.
- *maxInArr* and *minInArr* implement linear search procedures for largest and smallest, respectively, values in an array (taken from `http://proval.lri.fr/`). We check that the functions respectively correctly return the biggest or smallest value of the array.
- *nonDisj* is a procedure that takes in input an array a of integers and saves in a local array variable b all the position i where $a[i] > 0$, such that the property $a[b[j]] > 0$ is satisfied for all the element j

such that $b[j]$ is smaller than than the size of a. We check that this property is satisfied by every position of b that has been initialized by the procedure.

- *partition* implements an algorithm to distribute the content of an array in two: one holding all non-negative values and the other all the negative values (taken from [Hoder et al., 2010]). We check that the two target arrays contains only non-negative and positive values, respectively.
- *running* is the procedure in Figure 2.1. We check that assertion (2.8) is never violated.
- *vararg* is the procedure in [Hoder et al., 2010] searching for the first position of the input array storing the symbolic constant *NULL*, marking the point up to which the array has been initialized. We check that the procedure returns the first position where the input array contains the value *NULL*.

To quantitatively characterize the problems in the two benchmark suites, we have identified the following three parameters: the numbers L and N of non-nested and nested, respectively, loops in the body of the program and the number Q of quantifiers in the safety property. The interest of these figures lies in the fact that SAFARI, like any tool based on a CEGAR-like strategy, suffers from

- the presence of several non-nested loops in the program. This is because each counter-example found by unwinding must go through the L loops. Thus, refinement should be able to generalize the invariants for all the L loops from the same (inconsistent) formula representing the (infeasible) counter-example. In this respect, the problems identified by "copy$N$," where N represents the number of loops in the program, in SUITE 2 are particularly relevant (notice that L = N).

- the "depth" N of nested loops[3]. The problem is that the infeasibility of a counter-example may derive from the interaction of variables that are updated in two or more nested loops. For example, in the case of two nested loop, the behavior of the inner loop is influenced by the operations performed in the outer loop. The interplay among the variables is

[3] N = 0 means that the program does not have nested loops, N = 1 identifies programs with at least one nested loop, etc.

indeed reflected in the counter-example found by unwinding and refinement must then be able to synthesize an invariant describing the possibly complex relationships among the elements stored in several array variables. In this respect, the problems *binarySort* and *bubbleSort* in SUITE 2 are particularly interesting because they contain two nested loops (N = 1).

- the presence of a number Q of quantifiers in the property to be verified. The crucial observation here is that unbounded arrays (i.e. of finite but unknown dimension) require the capability to identify quantified predicates for synthesizing the invariants for discharging the safety property. So, the higher the number Q of quantified variables in the property, the higher the complexity to find quantified predicates that imply the property. In this respect, the problems identified by "QDn" in SUITE 1 are particularly relevant (notice that Q = 1). In fact, comparing the performances of SAFARI on "Dn" and "QDn" will give an idea of the advantages and disadvantages to use properties expressed by quantified (Q > 0) and quantifier-free (Q = 0) assertions, respectively.

4.2.2 Importance of the heuristics

We now show that the heuristics *Term Abstraction* (TA) and *Counterexample Minimization* (CM) – described in section 4.1.1 and section 4.1.2, respectively – are key to the scalability of SAFARI. To show this, we have run SAFARI on both benchmark suites with the heuristics turned on and off. All the experiments have been conducted on a computer equipped with an Intel(R) Core(TM)2 Quad CPU @ 3.00GHz and 12 GB of RAM running Linux Debian "jessie." The complete benchmark suites and the executable of SAFARI used for the evaluation are available at `http://verify.inf.usi.ch/content/safari`. The results are reported in Table 4.1 and 4.2 for SUITE 1 and Table 4.3 and 4.4 for SUITE 2.

In all the tables, the column 'PB.' reports the identifier of the problem together with the tuple (L, N, Q) representing the number of loops, maximum level of nesting, and number of quantified variables in the assertions, respectively (see section 4.2.1 for a description). Since SUITE 1 contains both safe and unsafe problems, the column 'STATUS' of Table 4.1 and 4.2 reports if the problem is safe or unsafe. Since SUITE 2 contains a safe and an unsafe version of the same problem, Table 4.3 and 4.4 groups the statistics of SAFARI for the safe and unsafe variants of the same problem. Table 4.1 and Table 4.3

report the execution time of SAFARI (in seconds with a time out of 1 hour), Table 4.2 and 4.4 report the number of refinements (with a maximum of 150) used by SAFARI. Each table reports measures (time or number of refinements) for the following configurations of SAFARI: no use of abstraction (NOA), i.e. SAFARI performs backward reachability, use of abstraction with both heuristics switched off (NOH), use of abstraction with only Counter-example Minimization turned on (CM), use of abstraction with only Term Abstraction turned on (TA), use of abstraction with both heuristics turned on (CMTA).

The results reported in the tables show the importance of heuristics for the scalability of SAFARI. Heuristics play a crucial role in allowing SAFARI to convergence on safe programs: without them, in fact, SAFARI is almost never able to converge as shown by looking at the columns NOH in all the tables. We also observe that the role of the two heuristics is quite different. In fact, Counter-example Minimization alone allows SAFARI to converge on few more examples than when the tool is executed without options (compare the columns NOH and CM in the tables). Instead, Term Abstraction alone enables SAFARI to converge on many more problems (compare the columns NOH and TA in the tables). The problems on which SAFARI fails to converge with Term Abstraction only turned on are successfully verified by using both heuristics (compare the columns TA and CMTA in the tables). We can explain the differences in the impact of the heuristics as follows.

Recall from section 4.1.1 that Term Abstraction allows SAFARI to induce the interpolation procedure to return an interpolant that could be potentially more useful for refinement. In other words, Term Abstraction has an impact on *how* a counter-example is refined. Instead, Counterexample Minimization (recall section 4.1.2) tries to find the smallest unsatisfiable suffix of the counter-example in order to prune the search space as much as possible. In other words, Counterexample Minimization addresses the problem to find *where* to refine a counter-example. So, Term Abstraction alone is sufficient when the counter-examples to be refined are not long and it is thus crucial how refinement is performed. When counter-examples become longer, it is also important where to refine them, not only how. On such problems, it is only the combination of the two heuristics that is winning.

We conclude by observing that in case of unsafe problems, the overhead of using abstractions with the heuristics turned on is small (compare the columns NOA and CMTA in the tables for unsafe problems).

4.3 Discussion

We can summarize the findings of the experimental analysis as follows.

The success of SAFARI is determined by a careful tuning of precision in the refinement phase of the CEGAR loop on which the tool is based. In particular, Term Abstraction is capable of inducing the interpolation procedure to provide the "right" interpolants, i.e. formulæ that give rise to a more precise but not too precise abstraction of the program so as to permit SAFARI to converge. When counter-examples are longer, the use of Counter-example Minimization in conjunction with Term Abstraction becomes crucial to drive the refinement procedure towards a good and successful refinement of the abstract model.

The capability to specify quantified assertions and reasoning about arrays of unbounded length allows SAFARI to consider compact annotations and verify programs regardless of the number of cells in an array. This makes the results of the verification more useful since safety holds for arrays of finite but arbitrary size and, at the same time, may improve performances by using compact (symbolic) representations of the set of (backward) reachable states during unwinding.

The experimental evaluation of the next chapter will build on the framework depicted in Figure 4.1. Chapter 8 presents a thorough experimental evaluation of BOOSTER, a framework integrating the techniques presented so far with other static analysis solutions that will be presented along the thesis. BOOSTER will be compared with other relevant state-of-the-art tools.

The following chapters will describe orthogonal techniques with respect to abstraction that will allow for an effective analysis of programs with arrays. As shown in this chapter, abstraction-based solutions do suffer from a degree of randomness requiring several heuristics. To go beyond the limits of abstraction-based frameworks, one has to combine them with different solutions, one of which is *acceleration*. The combination of abstraction and acceleration is what will determine the real practical effectiveness of BOOSTER.

4.4 Related work

In this section we present different tools that are related to SAFARI.

The tool ACSAR [Seghir et al., 2009] is a software model-checker adopting a backward reachability procedure in which new predicates are generated by simulating the "pre" operator on spurious counterexamples. This constitutes the main difference with respect to our approach, which performs refinement

by means of interpolants. Another related tool is EUREKA [Armando et al., 2007b, Armando et al., 2007a] implementing, as discussed earlier in section 3.5, an abstraction-refinement procedure for linear programs with *fixed-size* arrays.

The ICE framework [Garg et al., 2014] targets the problem to generate quantified safe inductive invariants by exploiting some machine learning techniques.

The tools ASTREE and CLOUSOT implement some solutions for the analysis of programs with arrays as described respectively in [Blanchet et al., 2002] and [Cousot et al., 2011]. The problem here is that the application of the join and widening operators (the last one required for ensuring the termination of the analysis) cause a loss of precision resulting in different false alarms. From an evaluation of the on-line version of CLOUSOT, available at , we observed the following results. On the safe versions of the 25 programs in SUITE 2, CLOUSOT is able to verify only 4 programs (namely, *find*, *init*, *partition*, and *vararg*) while on the unsafe versions is able to identify the bug for 2 programs only (namely, *partition* and *vararg*). This confirms our intuition that the trade-off between precision and efficiency in CLOUSOT is not satisfactory when (quantified) assertions about array programs are to be verified.

The VAMPIRE theorem prover is the only theorem prover, to the best of our knowledge, with interpolation features [Hoder et al., 2011]. It has been exploited inside LINGVA [Dragan and Kovács, 2014], a tool for the analysis of `C` programs.

PREDATOR [Dudka et al., 2011, Dudka et al., 2013] is another well-known software model-checker targeting shape analysis and verification of code manipulating of dynamic data-structures. While PREDATOR was successfully used to prove memory safety of programs operating on unbounded linked lists [Beyer, 2013], it is not yet able to prove that the array returned by a sorting algorithm is sorted. Additionally, the abstraction algorithms implemented in PREDATOR cannot handle arrays of unbounded size. However, as pointed out in [Dillig et al., 2010], the two techniques are orthogonal and their integration is likely to benefit both of them.

4.5 Summary

In this chapter we presented an effective implementation of the results presented in chapter 3. We discussed the necessary heuristics implemented to make the tool effective, both on the side of handling quantifiers and on the side of driving the interpolation procedure towards the generation of good in-

terpolants.

The most important heuristic of SAFARI is *Term Abstraction* (section 4.1). It allows to generalize interpolants, increasing the chances of convergence of the model-checker. SAFARI couples Term Abstraction with another heuristics called *Counterexample Minimization*. Counterexample Minimization allows to preserve as much as possible the status of the state-space explored by the tool.

The tool has been tested on various examples taken from the recent literature on invariant generation. The experiments show the importance of the heuristics we implemented and the effectiveness of the tool (section 4.2).

Executables of SAFARI, the benchmark suite and some tutorials are available from `http://verify.inf.usi.ch/safari`.

4.5.1 Related publications

The results reported in this chapter have been published in the following papers:

- F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: SMT-Based Abstraction for Arrays with Interpolants. In P. Madhusudan and S.A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*. Springer, 2012.

- F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. An extension of lazy abstraction with interpolation for programs with arrays. *Formal Methods in System Design*, 45(1):63–109, 2014.

Pb. (l,n,q)	status	NoA	NoH	cm	ta	cmta
Timings [Time out = 3600] (in seconds)						
D01 (2,0,0)	safe	x	-	-	0.36	0.38
D02 (2,0,0)	safe	x	-	-	0.39	0.28
D03 (2,0,0)	safe	x	-	-	0.37	0.52
D04 (2,0,0)	unsafe	3.92	0.51	0.30	0.18	0.28
D06 (2,0,0)	unsafe	x	-	-	2.68	0.78
D08 (2,0,0)	safe	x	-	-	0.36	0.50
D09 (2,0,0)	safe	x	-	-	0.50	0.40
D11 (2,0,0)	unsafe	1.54	0.35	0.28	1.53	1.02
D13 (2,0,0)	unsafe	0.45	0.42	0.34	0.33	0.45
D14† (4,0,0)	safe	x	-	-	1.60	1.06
D15 (4,0,0)	unsafe	2.62	1.60	1.33	1.46	1.56
D16† (5,0,0)	safe	x	-	-	2.22	1.10
D17 (2,0,0)	safe	x	0.72	0.80	x	0.68
D20 (2,0,0)	safe	x	-	-	0.81	0.47
QD01 (1,0,1)	safe	x	x	x	0.38	0.39
QD02 (1,0,1)	safe	x	x	x	0.43	0.35
QD03 (1,0,1)	safe	x	x	x	0.36	0.38
QD04 (1,0,1)	unsafe	0.34	0	1.44	0.31	0.37
QD08 (1,0,1)	safe	x	x	x	0.36	0.21
QD09 (1,0,1)	safe	x	x	x	0.43	0.44
QD11 (1,0,1)	unsafe	0.46	0	0.36	0.63	0.58
QD13 (2,0,2)	unsafe	0.44	0	0.41	0.61	0.35
QD14† (3,0,1)	safe	x	x	x	0.78	0.64
QD15 (3,0,1)	unsafe	0.53	4	2.62	1.09	0.94
QD16† (4,0,1)	safe	x	-	-	1.38	1.14
QD20 (1,0,1)	safe	x	x	x	0.37	0.28

Table 4.1. Experiments on Suite 1: running time for SAFARI with different heuristics turned on and off. A 'x' indicates that SAFARI was not able to converge in the given time out of 1 hour. A '-' indicates that SAFARI was not able to converge with less than 150 refinements. The examples labeled with † have been pre-processed with *loop fusion*, a compiler optimization technique which replaces multiple loops (iterating over the same range) with a single one when the instructions in the body of a loop do not interfere with those in the bodies of the others (see, e.g., [Aho et al., 2007]).

Pb. (l,n,q)	status	NoA	NoH	cm	ta	cmta
Number of refinements [Maximum = 150]						
D01 (2,0,0)	safe	x	-	-	5	3
D02 (2,0,0)	safe	x	-	-	5	3
D03 (2,0,0)	safe	x	-	-	5	3
D04 (2,0,0)	unsafe	0	0	0	0	0
D06 (2,0,0)	unsafe	x	-	-	2	2
D08 (2,0,0)	safe	x	-	-	5	3
D09 (2,0,0)	safe	x	-	-	5	3
D11 (2,0,0)	unsafe	0	0	0	0	0
D13 (2,0,0)	unsafe	0	0	0	0	0
D14† (4,0,0)	safe	x	-	-	8	8
D15 (4,0,0)	unsafe	0	6	4	9	9
D16† (5,0,0)	safe	x	-	-	18	14
D17 (2,0,0)	safe	x	3	3	x	4
D20 (2,0,0)	safe	x	-	-	5	3
QD01 (1,0,1)	safe	x	x	x	2	2
QD02 (1,0,1)	safe	x	x	x	2	2
QD03 (1,0,1)	safe	x	x	x	2	2
QD04 (1,0,1)	unsafe	0	0	0	0	0
QD08 (1,0,1)	safe	x	x	x	2	2
QD09 (1,0,1)	safe	x	x	x	2	2
QD11 (1,0,1)	unsafe	0	0	0	0	0
QD13 (2,0,2)	unsafe	0	0	0	0	0
QD14† (3,0,1)	safe	x	x	x	6	6
QD15 (3,0,1)	unsafe	0	4	3	7	7
QD16† (4,0,1)	safe	x	-	-	12	12
QD20 (1,0,1)	safe	x	x	x	2	2

Table 4.2. Experiments on Suite 1: number of refinements required by safari with different heuristics turned on and off. A 'x' indicates that safari was not able to converge in the given time out of 1 hour. A '-' indicates that safari was not able to converge with less than 150 refinements. The examples labeled with † have been pre-processed with *loop fusion*, a compiler optimization technique which replaces multiple loops (iterating over the same range) with a single one when the instructions in the body of a loop do not interfere with those in the bodies of the others (see, e.g., [Aho et al., 2007]).

	SAFE				UNSAFE				
PB. (L,N,Q)	NOH	CM	TA	CMTA	NOA	NOH	CM	TA	CMTA
TIMINGS [TIME OUT = 3600] (IN SECONDS)									
binarySort (3,1,2)	-	0.93	4.20	2.81	3.95	27.22	-	8.26	6.53
bubbleSort (2,1,2)	-	-	1.20	0.97	1.04	14.73	13.84	8.89	8.26
comp (1,0,1)	x	x	0.25	0.40	0.32	0.36	0.39	0.34	0.40
compM (1,0,1)	x	x	0.67	0.53	0.38	0.49	0.58	0.36	0.48
copy (1,0,1)	x	x	1.58	0.23	0.28	0.29	0.41	0.19	0.34
copy2 (2,0,1)	-	-	x	0.61	0.33	0.44	0.49	0.33	0.45
copy3 (3,0,1)	-	-	x	1.02	0.39	0.57	0.67	0.51	0.57
copy4 (4,0,1)	-	-	x	1.77	0.45	0.89	0.88	0.64	0.78
copy5 (5,0,1)	-	-	x	3.47	0.50	1.19	1.15	0.83	0.98
copy6 (6,0,1)	-	-	x	6.73	0.57	1.56	1.52	1.20	1.22
copy7 (7,0,1)	-	-	x	9.27	0.64	2.13	1.93	1.23	1.51
copy8 (8,0,1)	-	-	x	15.89	0.67	2.81	2.48	1.40	1.76
copy9 (9,0,1)	-	-	x	24.84	0.72	3.36	3.14	1.71	2.17
copy10 (10,0,1)	-	-	x	36.45	0.80	4.84	3.92	2.59	2.57
find (1,0,1)	x	x	0.42	0.60	0.28	0.23	0.34	0.21	0.36
findTest (2,0,0)	x	-	1.33	1.22	0.41	0.63	1.36	0.59	0.85
heapArr (1,0,0)	5.56	3.85	0.80	0.88	0.34	0.75	0.85	0.31	0.51
init (1,0,1)	x	x	0.37	0.30	0.29	0.17	0.28	0.17	0.31
initTest (2,0,0)	-	-	x	1.53	0.35	0.40	0.54	0.26	0.42
maxInArr (1,0,1)	-	-	0.43	0.30	0.29	0.29	0.42	0.23	0.38
minInArr (1,0,1)	-	-	0.43	0.46	0.29	0.30	0.42	0.23	0.39
nonDisj (1,0,2)	-	-	0.60	0.70	0.55	0.59	0.69	0.54	0.76
partition (1,0,1)	x	x	0.48	0.53	2.24	1.81	1.86	0.38	0.61
running (2,0,0)	x	x	0.92	0.87	0.28	0.44	0.47	0.29	0.46
vararg (1,0,1)	x	x	0.44	0.46	0.19	0.27	0.30	0.21	0.35

Table 4.3. Experiments on SUITE 2: running time for SAFARI with different heuristics turned on and off. A 'x' indicates that SAFARI was not able to converge in the given time out of 1 hour. A '-' indicates that SAFARI was not able to converge in less than 150 refinements. We do not report the column NOA for safe problems since SAFARI always diverges on them when abstraction is disabled.

	SAFE				UNSAFE				
PB. (L,N,Q)	NOH	CM	TA	CMTA	NOA	NOH	CM	TA	CMTA
NUMBER OF REFINEMENTS [MAXIMUM = 150]									
binarySort (3,1,2)	-	7	21	21	0	61	-	6	6
bubbleSort (2,1,2)	-	-	5	5	0	39	39	14	14
comp (1,0,1)	x	x	2	2	0	1	1	1	1
compM (1,0,1)	x	x	4	4	0	3	3	2	2
copy (1,0,1)	x	x	2	2	0	2	2	1	1
copy2 (2,0,1)	-	-	x	6	0	2	2	2	2
copy3 (3,0,1)	-	-	x	12	0	3	3	3	3
copy4 (4,0,1)	-	-	x	20	0	4	4	4	4
copy5 (5,0,1)	-	-	x	30	0	5	5	5	5
copy6 (6,0,1)	-	-	x	42	0	6	6	6	6
copy7 (7,0,1)	-	-	x	56	0	7	7	7	7
copy8 (8,0,1)	-	-	x	72	0	8	8	8	8
copy9 (9,0,1)	-	-	x	90	0	9	9	9	9
copy10 (10,0,1)	-	-	x	110	0	10	10	10	10
find (1,0,1)	x	x	3	4	0	1	1	1	1
findTest (2,0,0)	x	-	14	19	0	6	13	8	8
heapArr (1,0,0)	68	54	9	9	0	9	9	4	4
init (1,0,1)	x	x	2	2	0	0	0	0	0
initTest (2,0,0)	-	-	x	11	0	3	3	1	1
maxInArr (1,0,1)	-	-	3	3	0	2	2	2	2
minInArr (1,0,1)	-	-	3	3	0	2	2	2	2
nonDisj (1,0,2)	-	-	0	0	0	4	4	5	5
partition (1,0,1)	x	x	1	1	0	7	7	2	2
running (2,0,0)	x	x	6	10	0	2	2	3	3
vararg (1,0,1)	x	x	4	4	0	1	1	2	2

Table 4.4. Experiments on SUITE 2: number of refinements required by SAFARI with different heuristics turned on and off. A 'x' indicates that SAFARI was not able to converge in the given time out of 1 hour. A '-' indicates that SAFARI was not able to converge in less than 150 refinements. We do not report the column NOA for safe problems since SAFARI always diverges on them when abstraction is disabled.

Chapter 5

Acceleration techniques for relations over arrays

This chapter is devoted to the presentation of techniques for computing the *acceleration* of transition relations with arrays. Acceleration is a well-known approach in model-checking. It requires to compute the transitive closure of relations expressing the system evolution and it is exploited to compute *precisely* the set of reachable states of a transition system. That is, given a transition system $\mathcal{S}_{\mathcal{T}} = (\mathbf{v}, l_{\mathsf{init}}, l_{\mathsf{error}}, T)$ and the acceleration of T, usually denoted with T^*, we can compute the precise set of states reachable by $\mathcal{S}_{\mathcal{T}}$. This is represented by the post-image of l_{init} with respect to T^*. Once this formula, say $R(\mathbf{v})$, has been computed we can check whether $R(\mathbf{v})$ intersects with the error location and infer the safety of $\mathcal{S}_{\mathcal{T}}$. This check can be performed by testing the $\mathcal{T}$-satisfiability of $R(\mathbf{v}) \wedge pc = l_{\mathsf{error}}$.

This strategy for solving the reachability analysis problem may work for a class of systems with integer variables, as shown for example in [Bozga et al., 2014], but does not work in general for those handling arrays. The reason is the following. Transitive closure is a logical construct that is far beyond first order logic: either infinite disjunctions or higher order quantifiers or, at least, fixpoint operators are required to express it. Indeed, due to the compactness of first order logic, transitive closure (even modulo the axioms of a first order theory) is first-order definable only in trivial cases. For expressive formalisms like the aforementioned ones, it is problematic to find efficient solvers (if any at all), which can be used in verification. This implies that even if we would be able to express T^*, the safety test $R(\mathbf{v}) \wedge pc = l_{\mathsf{error}}$ would be undecidable, nullifying the benefits of acceleration.

To address this problem, it is required first to identify constrained classes

of relations, from a syntactic point of view, admitting a definable transitive closure within a suitable first-order theory. The approach of [Bozga et al., 2014], for example, builds on the fact that loops represented by (conjunction of) octagonal relations, i.e., relations of the kind $\pm x \pm y \leq c$ have definable acceleration within $\mathcal{LIA}$ [Bozga et al., 2009a]. This fact combined with some requirements on the control-flow structure of the program allow to show that the reachability problem for a class of programs with integer variables is decidable.

Recall that in this thesis we defined a theory as a pair made by a signature Σ and a class of Σ-structures $\mathcal{C}$ (see Definition 2.1.8). Such definition is different from the classical one where a theory is identified as a set of axioms (see, e.g., [Mendelson, 1997]). By taking a theory as a class of structures the property of compactness fails, and it might well happen that transitive closure becomes first-order definable (the first order definition being valid just inside the class $\mathcal{C}$ - which is often reduced to a single structure).

The contributions of this chapter are the following:

- We show that inside the combined theory of Presburger arithmetic augmented with free function symbols, the acceleration of some classes of relations – corresponding, in our application domain, to relations involving arrays and counters – can be expressed in first order language. This result comes at a price to allow nested quantifiers.

- The nested quantification introduced by the acceleration procedure can be problematic in practical applications. To address this complication we show how to take care of the quantifiers added by the acceleration procedure: the idea is to import in this setting the so-called *monotonic abstraction* technique [Abdulla et al., 2007b, Abdulla et al., 2007a]. Such technique has been reinterpreted and analyzed in a declarative context in [Alberti et al., 2012d]: from a logical point of view, it amounts to a restricted form of *instantiation for universal quantifiers*.

- We show that acceleration can be effectively used to check the safety of programs with arrays. In particular, as discussed in Chapter 3, one of the biggest problems in verifying safety properties of array programs is designing procedures for the synthesis of relevant quantified predicates. In typical sequential programs, the guarded assignments used to model the program instructions are ground and, as a consequence, the formulæ representing backward reachable states are ground too. However, the

invariants required to certify the safety of such programs contain quantifiers. Our acceleration procedure is able to supply the required quantified predicates.

We also conjecture that acceleration and abstraction are orthogonal techniques, in the sense that they offers two different ways for achieving the same goal. Having different strengths and weaknesses, their combination likely lead to a framework overcoming their individual limitations.

5.1 SMT-based backward reachability

We assume the notions introduced in section 2.3, and fix $\mathcal{T}$ to be the theory of array obtained by enriching the signature of $\mathcal{LIA}$ with free function symbols and free constants. As a consequence, when we speak about validity or satisfiability of a formula, we mean satisfiability and validity in all structures having the *standard* structure of natural numbers as reduct.

5.1.1 Backward reachability

Backward reachability is a standard procedure for checking the safety of a transition system $\mathcal{S}_\mathcal{T}$. As we did in chapter 3, we are now presenting a backward reachability procedure that will be used in the remaining part of the chapter. The procedure is given in Figure 5.1. It is fed with a transition system $\mathcal{S}_\mathcal{T} = (\mathbf{v}, l_{\mathsf{init}}, l_{\mathsf{error}}, T)$, as defined in section 2.3. It explores, through symbolic representation, all states leading to the error location l_{error} in one step, then in two steps, in three steps, etc. until either we find a fixpoint or until we reach l_{init}. Similarly to the solution presented in chapter 3, it is convenient to arrange the explored state-space in a tree: the tree has arcs labeled by transitions and nodes labeled by formulæ over $\mathbf{v}$. Leaves of the tree are labeled as 'checked', 'unchecked' or 'covered'. The tree is built according to the rules of the BReach procedure of Figure 5.1.

Termination of BReach is triggered by two events. The first arises if the safety test succeeds. In this case the transition system $\mathcal{S}_\mathcal{T}$ is unsafe. In the second case, all the leaves are flagged as 'covered'. This happens because at some point, all the nodes generated are labeled with formulæ that are covered by the other nodes. As discussed in section 2.2.1, the satisfiability of the safety and fixpoint check depends on the shape (e.g., presence of quantifiers) of the checked formulæ. The presence of formulæ falling outside decidable classes in the fixpoint test might prevent the termination of the algorithm when executed

BREACH

INITIALIZATION: a single node tree labeled by $pc = l_{\mathsf{error}}$ and is marked 'unchecked'.

CHECK: pick an unchecked leaf L labeled with K. If $K \wedge pc = l_{\mathsf{init}}$ is satisfiable ('safety test'), exit and return unsafe. If it is not satisfiable, check whether there is a set C of uncovered nodes such that (i) $L \notin C$ and (ii) K is inconsistent with the conjunction of the negations of the formulæ labeling the nodes in C ('fixpoint check'). If it is so, mark L as 'covered' (by C). Otherwise, mark L as 'checked'.

EXPANSION: pick a checked leaf L labeled with K. For each transition $\tau_i \in T$, add a new leaf below L labeled with $Pre(\tau_i, L)$ and marked as 'unchecked'. The arc between L and the new leaf is labeled with τ_i.

SAFETY EXIT: if all leaves are covered, exit and return safe.

Figure 5.1. The BREACH backward reachability procedure.

on safe transition systems. Not being able to check the safety test, instead, might compromise the ability of BREACH to check even the unsafety of any transition system, turning BREACH into a useless procedure.

When acceleration comes into play, this last scenario becomes feasible. This means that suitable countermeasures have to be taken in order to allow at least sound solutions.

5.1.2 Classification of sentences and transitions

BREACH represents the set of the backward reachable configurations of $\mathcal{S}_{\mathcal{T}} = (\mathbf{v}, l_{\mathsf{init}}, l_{\mathsf{error}}, T)$ with formulæ. Recall from section 2.3 that $\mathbf{v} = \langle \mathbf{a}, \mathbf{s}, pc \rangle$ where $\mathbf{a}$ is a tuple of array variables and $\mathbf{s}$ a tuple of scalar variables. We classify such formulæ into three classes:

- *ground* sentences, i.e., sentences of the kind $\phi(\mathbf{v})$;
- Σ_1^0-*sentences*, i.e., sentences of the form $\exists \underline{i}.\ \phi(\underline{i}, \mathbf{v})$;
- Σ_2^0-*sentences*, i.e., sentences of the form $\exists \underline{i} \forall \underline{j}.\ \phi(\underline{i}, \underline{j}, \mathbf{v})$.

We remark that in our context satisfiability can be fully decided only for ground sentences and Σ_1^0-sentences (see the results in section 2.2.1), while only

subclasses of Σ_2^0-sentences admit a decision procedure, e.g., those discussed in [Bradley et al., 2006, Ge and de Moura, 2009].

A classification of transition formulæ will also be needed in this chapter. We classify transition formulæ in three groups:

- *ground assignments*, i.e., transitions of the form

$$pc = l \ \wedge \ \phi_L(\mathbf{s}, \mathbf{a}) \ \wedge \ pc' = l' \ \wedge \ \mathbf{a}' = \lambda j.\, G(\mathbf{s}, \mathbf{a}, j) \ \wedge \ \mathbf{s}' = H(\mathbf{s}, \mathbf{a}) \quad (5.1)$$

- Σ_1^0-*assignments*, i.e., transitions of the form

$$\exists \underline{k} \left(\begin{array}{c} pc = l \ \wedge \ \phi_L(\mathbf{s}, \mathbf{a}, \underline{k}) \ \wedge \ pc' = l' \ \wedge \\ \mathbf{a}' = \lambda j.\, G(\mathbf{s}, \mathbf{a}, \underline{k}, j) \ \wedge \ \mathbf{s}' = H(\mathbf{s}, \mathbf{a}, \underline{k}) \end{array} \right) \quad (5.2)$$

- Σ_2^0-*assignments*, i.e., transitions of the form

$$\exists \underline{k} \left(\begin{array}{c} pc = l \ \wedge \ \phi_L(\mathbf{s}, \mathbf{a}, \underline{k}) \ \wedge \ \forall \underline{j} \ \psi_U(\mathbf{s}, \mathbf{a}, \underline{k}, \underline{j}) \ \wedge \\ pc' = l' \ \wedge \ \mathbf{a}' = \lambda j.\, G(\mathbf{s}, \mathbf{a}, \underline{k}, j) \wedge \ \mathbf{s}' = H(\mathbf{s}, \mathbf{a}, \underline{k}) \end{array} \right) \quad (5.3)$$

where, as usual, $G = G_1, \ldots, G_s$, $H = H_1, \ldots, H_t$ are tuples of case-defined functions.

Definition 5.1.1 (Acceleration)**.** *The* composition $\tau_1 \circ \tau_2$ *of two transitions* $\tau_1(\mathbf{v}, \mathbf{v}')$ *and* $\tau_2(\mathbf{v}, \mathbf{v}')$ *is expressed by the formula* $\exists \mathbf{v}_1(\tau_1(\mathbf{v}, \mathbf{v}_1) \wedge \tau_2(\mathbf{v}_1, \mathbf{v}'))$. *The* n*-th composition of a transition* $\tau(\mathbf{v}, \mathbf{v}')$ *with itself is recursively defined by* $\tau^1 := \tau$ *and* $\tau^{n+1} := \tau \circ \tau^n$. *The* acceleration *of* τ *is* $\bigvee_{n \geq 1} \tau^n$.

We also recall the definition of preimage: the preimage of a formula $\alpha(\mathbf{v})$ with respect to a transition $\tau(\mathbf{v}, \mathbf{v}')$ is represented by the formula

$$Pre(\tau, \alpha) \ \equiv \ \exists \mathbf{v}'.\, (\tau(\mathbf{v}, \mathbf{v}') \wedge \alpha(\mathbf{v}'))\,. \quad (3.2)$$

The following proposition is proved by straightforward syntactic manipulations:

Proposition 5.1.1. *Let* τ, τ_1, τ_2 *be transition formulæ and let* $K(\mathbf{v})$ *be a formula. We have that:* (i) *if* τ_1, τ_2, τ, K *are ground, then* $\tau_1 \circ \tau_2$ *is a ground assignment and* $Pre(\tau, K)$ *is a ground formula;* (ii) *if* τ_1, τ_2, τ, K *are* Σ_1^0*, then* $\tau_1 \circ \tau_2$ *is a* Σ_1^0*-assignment and* $Pre(\tau, K)$ *is a* Σ_1^0*-sentence;* (iii) *if* τ_1, τ_2, τ, K *are* Σ_2^0*, then* $\tau_1 \circ \tau_2$ *is a* Σ_2^0*-assignment and* $Pre(\tau, K)$ *is a* Σ_2^0*-sentence.*

```
procedure Reverse ( I[N + 1]; O[N + 1] ){
    c = 0;
    while (c ≠ N + 1) {
      O[c] = I[N − c];
      c = c + 1;
    }
    assert (∀x ≥ 0, y ≥ 0( x + y = N → I[x] = O[y] ) )
}
```

Figure 5.2. A function for reversing the elements of an array I into another array O.

In our application domain, transitions are generally ground assignments or, at most, Σ_1^0-assignments, as a result of translating the violation of universally quantified assertions. In this case BREACH generates only Σ_1^0-formulæ and completeness of safety tests is guaranteed by the fact that satisfiability of Σ_1^0-formulæ is decidable via Skolemization and then the application of the results of section 2.2.1. From this consideration, the following fact holds:

Theorem 5.1.1. BREACH *is partially correct*[1] *for transition systems whose transitions are either ground assignments or* Σ_1^0*-assignments.*

For fixpoint tests the situation is different. The generation of Σ_1^0-formulæ during the backward reachability analysis implies that these tests reduce to the $\mathcal{T}$-satisfiability of Σ_2^0-formulæ. This problem has been already analyzed in section 4.1.3, where we identified several practical heuristics to tackle it. In any case, given the general undecidability of this class of formulæ (recall the discussion of section 2.2.2), the adoption of sound but incomplete algorithms may, therefore, compromise the termination of BREACH, but not correctness of the answer.

Divergence phenomena are not only caused by the incompleteness of fixpoint tests, though. In fact, divergence persists even in cases where fixpoint tests are precise, as we will show in the example below. One source of divergence is the fact that we are unable to compute "in one shot" the effect of

[1] *Partially correctness* means that when the procedure terminates it gives a correct information about the safety of the input array-based transition system.

executing finitely many times a given sequence of transitions. Acceleration can solve this problem.

Example 5.1.1. Consider the procedure `Reverse` of Figure 5.2. This procedure can be represented by the array-based transition system $\mathcal{S}_{\mathcal{T}_A} = (\mathbf{v}, l_0, l_3, T)$, where $\mathcal{T}_A$ is the combination of the theory $\mathtt{ARR}^1(\mathcal{LIA})$ to which we added a constant N, and a single-sorted enumerated data-type theory which unique sort is interpreted over $\{l_0, l_1, l_2, l_3\}$. $\mathbf{v} := \langle I, O, c, pc \rangle$. The sort of pc is interpreted over the set $\{l_0, l_1, l_2, l_3\}$. T contains the following transitions:

$$
\begin{aligned}
\tau_0(\mathbf{v}, \mathbf{v}') &\equiv\ pc = l_0 \wedge pc' = l_1 \wedge c' = 0 \\
\tau_1(\mathbf{v}, \mathbf{v}') &\equiv\ pc = l_1 \wedge c \neq N + 1 \wedge c' = c + 1 \wedge O' = \mathsf{store}(O, c, I(N - c)) \\
\tau_2(\mathbf{v}, \mathbf{v}') &\equiv\ pc = l_1 \wedge c = N + 1 \wedge pc' = l_2 \\
\tau_3(\mathbf{v}, \mathbf{v}') &\equiv\ pc = l_2 \wedge \exists z_1 \geq 0, z_2 \geq 0\ (z_1 + z_2 = N \wedge I(z_1) \neq O(z_2)\ ) \wedge pc' = l_3\ .
\end{aligned}
$$

Recall from section 2.2.3 that $\mathsf{store}(b, i, e)$ abbreviates the expression $\lambda j.\mathsf{if}\ (j = i)\ \mathsf{then}\ e\ \mathsf{else}\ b[j]$.

If we apply the BReach procedure on this example, we get an infinite labeled unwinding with a branch whose nodes - after routine simplifications - are labeled as follows:

$$
\begin{aligned}
(K)\ &\ pc = l_3 \\
(K')\ &\ pc = l_2 \wedge \exists z_1, z_2\ \psi(z_1, z_2) \\
(K'')\ &\ pc = l_1 \wedge \exists z_1, z_2\ \psi(z_1, z_2) \wedge c = N + 1 \\
(K_0)\ &\ pc = l_1 \wedge \exists z_1, z_2\ \psi(z_1, z_2) \wedge c = N \wedge z_2 \neq N \\
(K_1)\ &\ pc = l_1 \wedge \exists z_1, z_2\ \psi(z_1, z_2) \wedge c = N - 1 \wedge z_2 \neq N \wedge z_2 \neq N - 1 \\
&\ \dots \\
(K_i)\ &\ pc = l_1 \wedge \exists z_1, z_2\ \psi(z_1, z_2) \wedge c = N - i \wedge z_2 \neq N \wedge \dots \wedge z_2 \neq N - i \\
&\ \dots
\end{aligned}
$$

where $\psi(z_1, z_2)$ stands for $z_1 \geq 0 \wedge z_2 \geq 0 \wedge z_1 + z_2 = N \wedge I(z_1) \neq O(z_2)$. We can explain the divergence phenomenon as follows. BReach assumes, by "refutation", that the error location can be reached and tries to build a counterexample. Let x and y be the position of the array I and O such that $I[x] \neq O[y]$ and $x + y = N$. Unwinding the transition relation in a backward fashion results in checking all the positions of the array before x, starting from $z_0 = N, z_1 = N - 1, z_2 = N - 2, \dots, z_k = N - k, \dots$. Such procedure never terminates since "we can always add" one position between the last z_k checked

and x.

5.2 Definability of Accelerated Relations

Acceleration can be of great help in limiting divergence of reachability analysis. As stated in Definition 5.1.1, acceleration can be expressed by using infinite disjunctions. In this section we investigate the existence of a class of relations over arrays for which the infinite disjunction is equivalent, modulo $\mathcal{T}$, to a first-order formula built over the signature of $\mathcal{T}$. That is, we want to characterize a class of τ's that is both of practical use and for which it is effectively computable a transition τ^+ such that

$$\mathcal{T} \models \forall \mathbf{v}, \mathbf{v}'.\tau^+(\mathbf{v}, \mathbf{v}') \;\leftrightarrow\; \bigvee_{n \geq 1} \tau^n(\mathbf{v}, \mathbf{v}')$$

Recall that in this thesis we defined a theory $\mathcal{T}$ as a pair $(\Sigma, \mathcal{C})$ comprising a signature and a class of Σ-structures, the models of $\mathcal{T}$. This definition is *not* equivalent to the one given in standard textbooks (e.g., [Mendelson, 1997]). Indeed, by taking a theory as a set of first-order sentences (the axioms of the theory), compactness holds, and by consequence the following fact:

Theorem 5.2.1. *Let Σ be a signature, $\mathcal{T}$ be a set of Σ-sentences and ϕ, ψ_n some Σ-sentences. Then,*

$$\mathcal{T} \models \phi \leftrightarrow \bigvee_{n \geq 0} \psi_n$$

iff exists an N such that

$$\mathcal{T} \models \phi \leftrightarrow \bigvee_{n \leq N} \psi_n$$

Proof. One side of the proof is trivial. For the other side, let us suppose that for any N, $\mathcal{T} \cup \{\neg\psi_n\}_{n \leq N}$ is consistent. By compactness, $\mathcal{T} \cup \{\neg\psi_n\}_{n \geq 0}$ is also consistent. It follows that $\mathcal{T} \not\models \phi \leftrightarrow \bigvee_{n \geq 0} \psi_n$ because there exists a model of $\mathcal{T}$ falsifying all the ψ_n. □

5.2.1 Iterators, selectors and local ground assignments

The first two ingredients we need to supply a useful format to compute accelerated transitions are *iterators* and *selectors*. The intuition here is that we need to model *how* the scalars used to index the arrays are (i) updated and (ii) used to index the arrays.

Iterators are meant to formalize the notion of a counter scanning the indexes of an array: the most simple iterators are increments and decrements, but one may also build more complex ones for different scans. We need to handle tuples of terms because we want to consider the case in which we deal with different arrays with possibly different scanning variables.

Given a m-tuple of terms

$$\mathbf{u}(\underline{x}) \ := \ u_1(x_1, \ldots, x_m), \ldots, u_m(x_1, \ldots, x_m) \tag{5.4}$$

containing the m variables $\underline{x} = x_1, \ldots, x_m$, we indicate with $\mathbf{u}^n$ the term expressing the n-times composition of (the function denoted by) $\mathbf{u}$ with itself. Formally, we have $\mathbf{u}^0(\underline{x}) := \underline{x}$ and

$$\mathbf{u}^{n+1}(\underline{x}) \ := \ u_1(\mathbf{u}^n(\underline{x})), \ldots, u_m(\mathbf{u}^n(\underline{x})) \ .$$

Definition 5.2.1 (Iterators)**.** *A tuple of terms* $\mathbf{u}$ *like* (5.4) *is said to be an* iterator *iff there exists an* m*-tuple of* $m+1$*-ary terms*

$$\mathbf{u}^*(\underline{x}, y) \ := \ u_1^*(x_1, \ldots, x_m, y), \ldots, u_m^*(x_1, \ldots, x_m, y) \tag{5.5}$$

such that for any natural number n *it happens that the formula*

$$\mathbf{u}^n(\underline{x}) = \mathbf{u}^*(\underline{x}, \bar{n}) \tag{5.6}$$

is valid.[2]

The second notion we need is that of *selectors*.

Definition 5.2.2 (Selectors)**.** *Given an iterator* $\mathbf{u}$*, we say that an* m*-ary term* $\kappa(x_1, \ldots, x_m)$ *is a* selector *for* $\mathbf{u}$ *iff there is an* $m+1$*-ary term* $\iota(x_1, \ldots, x_m, y)$ *yielding the validity of the formula*

$$z = \kappa(\mathbf{u}^*(\underline{x}, y)) \rightarrow y = \iota(\underline{x}, z) \ \ . \tag{5.7}$$

The term κ is a selector function that selects (and possibly modifies) one of the $\mathbf{u}$; in most applications (though not always) κ is a projection, represented as a variable x_i (for $1 \leq i \leq m$), so that $\kappa(\mathbf{u}^*(\underline{x}, y))$ is just the i-th component $u_i^*(\underline{x}, y)$ of the tuple of terms $\mathbf{u}^*(\underline{x}, y)$. In these cases, the formula (5.7) reads as

$$z = u_i^*(\underline{x}, y) \rightarrow y = \iota(\underline{x}, z) \ . \tag{5.8}$$

[2]Recall that $\bar{n}$ is the numeral of n, i.e. it is $s^n(0)$.

The meaning of condition (5.7) is that, once the input $\underline{x}$ and the selected output z are known, it is possible to identify uniquely (through ι) the number of iterations y that are needed to get z by applying κ to $\mathbf{u}^*(\underline{x}, y)$. That is, in order to compute the acceleration of a transition handling array variables we need to know whether a given cell can be reached by a scalar variable within a given number of iterations. The number $\iota(\underline{x}, z)$ gives "the only possible candidate" y number of iterations. $z = \kappa(\mathbf{u}^*(\underline{x}, y))$ checks if the candidate y is correct.

Example 5.2.1. The canonical example is when we have $m = 1$ and $\mathbf{u} := u_1(x_1) := x_1 + 1$; this is an iterator with $u_1^*(x_1, y) := x_1 + y$; as a selector, we can take $\kappa(x_1) := x_1$ and $\iota(x_1, z) := z - x_1$.

Example 5.2.2. The previous example can be modified, by choosing $\mathbf{u}$ to be $x_1 + \bar{n}$, for some integer $n \neq 0$: then we have $u^*(x_1, y) := x_1 + n \cdot y$, $\kappa(x_1) := x_1$, and $\iota(x_1, z) = (z - x_1)/n$ where / is integer division (recall that integer division by a given n is definable in Presburger arithmetic).

Example 5.2.3. If we move to more expressive arithmetic theories, like Primitive Recursive Arithmetic (where we have a symbol for every primitive recursive function), we can get much more examples. As an example with $m > 1$, we can take $\mathbf{u} := x_1 + x_2, x_2$ and get $u_1^*(x_1, x_2, y) = x_1 + y \cdot x_2$, $u_2^*(x_1, x_2, y) = x_2$. Here a selector is for instance $\kappa_1(x_1, x_2) := \bar{7} + x_1$, $\iota(x_1, x_2, z) := (z - x_1 - \bar{7})/x_2$. □

Example 5.2.4. Consider the loop

$$\mathsf{while}\ (\ \mathsf{true}\ )\ \{a[i] = e; i = i + 2;\}$$

For this loop, the iterator is $u(i) := i + 2$ and $u^*(i, y) = i + 2y$. Moreover $\kappa(x) := x$ and $\iota(i, z) := (z - i)/2$.

Suppose i takes the value 3 before entering the loop, and we want to check if $a[7]$ can be reached in at most 3 iterations. We can compute the result of $\iota(i, z) = (7 - 3)/2$, i.e., 2, and then check if in two iterations we actually reach $a[7]$ with the formula $u^*(i, 2) = 3 + 2 \cdot 2$. Given that $3 + 2 \cdot 2 = 7$, we know that in two iterations we reach position 7.

On the contrary, suppose that i starts from 3 and we want to check if $a[6]$ can be reached in at most 3 iterations. Once again we compute $\iota(i, z) = (6 - 3)/2$, from which we obtain that 1 is the candidate number of iterations that we have to reach $a[6]$. By checking the correctness of this result we obtain that $u^*(i, 1) = 3 + 2 \cdot 1 = 5$. This means that $a[6]$ cannot be reached if we start from $i = 3$.

5.2.2 Accelerating local ground assignments

Given an array-based transition system $\mathcal{S}_\mathcal{T}$ we can now look for conditions on transitions from T allowing to find their definable acceleration modulo $\mathcal{T}$.

Given an iterator $\mathbf{u}(\underline{x})$, a *selector assignment* for $\mathbf{a} := a_1, \ldots, a_r$ (relative to $\mathbf{u}$) is a tuple of selectors $\kappa := \kappa_1, \ldots, \kappa_r$ for $\mathbf{u}$.

Definition 5.2.3 (Purely arithmetical formulæ)**.** *A formula ψ (resp. a term t) is said to be* purely arithmetical *over a finite set of terms V iff it is obtained from a formula (resp. a term) not containing the extra free function symbols $\mathbf{a}, \mathbf{s}$ by replacing some free variables in it by terms from V.*

Let $\mathbf{v} = v_1, \ldots, v_r$ and $\mathbf{w} = w_1, \ldots, w_r$ be r-tuples of terms; below $\mathsf{store}(\mathbf{a}, \mathbf{v}, \mathbf{w})$ indicates the tuples $\mathsf{store}(a_1, v_1, w_1), \ldots, \mathsf{store}(a_r, v_r, w_r)$.

Definition 5.2.4 (Local ground assignments)**.** *A* local ground assignment *is a ground assignment of the form*

$$pc = l \ \wedge \ \phi_L(\mathbf{s}, \mathbf{a}) \ \wedge \ pc' = l \ \wedge \ \mathbf{a}' = \mathsf{store}(\mathbf{a}, \kappa(\tilde{\mathbf{s}}), \mathbf{t}(\mathbf{s}, \mathbf{a})) \ \wedge \ \tilde{\mathbf{s}}' = \mathbf{u}(\tilde{\mathbf{s}}) \ \wedge \mathbf{z}' = \mathbf{z} \tag{5.9}$$

where

(i) $\mathbf{s} = \tilde{\mathbf{s}}, \mathbf{z}$*;*

(ii) $\mathbf{u} = u_1, \ldots, u_{|\tilde{\mathbf{s}}|}$ *is an iterator;*

(iii) the terms κ are a selector assignment for $\mathbf{a}$ relative to $\mathbf{u}$;

(iv) the formula $\phi_L(\mathbf{s}, \mathbf{a})$ and the terms $\mathbf{t}(\mathbf{s}, \mathbf{a})$ are purely arithmetical over the set of terms $\{\mathbf{s}, \mathbf{a}(\kappa(\tilde{\mathbf{s}}))\} \cup \{a_i(z_j)\}_{1 \leq i \leq r, 1 \leq j \leq |\mathbf{z}|}$;

(v) the guard ϕ_L contains the conjuncts $\kappa_i(\tilde{\mathbf{s}}) \neq z_j$, for $1 \leq i \leq r$ and $1 \leq j \leq |\mathbf{z}|$.

Thus in a local ground assignment, there are various restrictions:

(a) the numerical variables are split into 'idle' variables $\mathbf{z}$ and variables $\tilde{\mathbf{s}}$ subject to update via an iterator $\mathbf{u}$;

(b) the program counter is not modified;

(c) the guard does not depend on the values of the a_i at cells different from $\kappa_i(\tilde{\mathbf{s}}), \mathbf{z}$;

(d) the update of the $\mathbf{a}$ are simultaneous writing operations modifying only the entries $\kappa(\tilde{\mathbf{s}})$.

Thus, the assignment is local and the relevant modifications it makes are determined by the selector locations. The 'idle' variables $\mathbf{z}$ are useful to accelerate branches of nested loops; the inequalities mentioned in (v) are automatically generated by making case distinctions in assignment guards.

Example 5.2.5. Consider again the procedure Reverse discussed in Example 5.1.1. The only candidate transition to be accelerated is τ_1, i.e.,

$$pc = l_1 \wedge c \neq N+1 \wedge pc' = 2 \wedge c' = c+1 \wedge I' = I \wedge O' = \mathsf{store}(O, c, I(N-c))$$

We have $\mathbf{z} = \emptyset$ and $\tilde{\mathbf{s}} = c$ and $\mathbf{a} = I, O$. The counter c is incremented by 1 at each application of τ_2. Thus, our iterator is $\mathbf{u} := x_1 + 1$ and the selector assignment assigns $\kappa_1 := N - x_1$ to I and $\kappa_2 := x_1$ to O. This way, I is modified (identically) at $N - c$ via $I' = \mathsf{store}(I, N-c, I(N-c))$ and O is modified at c via $O' = \mathsf{store}(O, c, I(N-c))$. The guard τ_2 is $c \neq N+1$. Since the formula $c \neq N+1$ and the term $I(N-c)$ are purely arithmetical over $\{c, I(N-c), O(c)\}$, we conclude that τ_1 is a local ground assignment.

Before showing that local ground assignments admit definable acceleration, we prove the following lemma.

Lemma 5.2.1. *If τ is a local ground assignment, τ^n can be expressed as follows*[3]

$$\bigwedge_{0 \leq k < n} \tilde{\phi}_L(\mathbf{u}^*(\tilde{\mathbf{s}}, \bar{k}), \mathbf{z}, \mathbf{a}(\kappa(\mathbf{u}^*(\tilde{\mathbf{s}}, \bar{k}))), \mathbf{a}[\mathbf{z}]) \wedge \tilde{\mathbf{s}}' = \mathbf{u}^*(\tilde{\mathbf{s}}, \bar{n}) \wedge \mathbf{a}' = \lambda j.\ F(\mathbf{s}, \mathbf{a}, \bar{n}, j) \tag{5.10}$$

where the tuple $F = F_1, \ldots, F_r$ of definable functions is given by

$$\begin{aligned} F_h(\mathbf{s}, \mathbf{a}, y, j) \ = \ & \texttt{if} \ \ 0 \leq \iota_h(\tilde{\mathbf{s}}, j) < y \ \wedge j = \kappa_h(\mathbf{u}^*(\mathbf{s}, \iota_h(\tilde{\mathbf{s}}, j))) \ \texttt{then} \\ & \tilde{t}_h(\mathbf{u}^*(\tilde{\mathbf{s}}, \iota_h(\tilde{\mathbf{s}}, j)), \mathbf{z}, \mathbf{a}(\kappa(\mathbf{u}^*(\tilde{\mathbf{s}}, \iota_h(\tilde{\mathbf{s}}, j)))), \mathbf{a}[\mathbf{z}]) \ \texttt{else} \ a_h[j] \end{aligned} \tag{5.11}$$

for $h = 1, \ldots, r$ (here $\iota_1, \ldots, \iota_r$ are the terms corresponding to $\kappa_1, \ldots, \kappa_r$ according to the definition of a selector for the iterator $\mathbf{u}$).

Proof. For $n = 1$, notice that $\tilde{\phi}_L(\mathbf{u}^*(\tilde{\mathbf{s}}, 0), \mathbf{z}, \mathbf{a}(\kappa(\mathbf{u}^*(\tilde{\mathbf{s}}, 0))), \mathbf{a}[\mathbf{z}])$ is equivalent to $\tilde{\phi}_L(\tilde{\mathbf{s}}, \mathbf{z}, \mathbf{a}(\kappa(\tilde{\mathbf{s}})), \mathbf{a}[\mathbf{z}])$, that $\tilde{\mathbf{s}}' = \mathbf{u}^*(\tilde{\mathbf{s}}, \bar{1})$ is equivalent to $\tilde{\mathbf{s}}' = \mathbf{u}(\tilde{\mathbf{s}})$ and

[3] We omit here and below the conjuncts $pc = l \wedge pc' = l \wedge \mathbf{z}' = \mathbf{z}$ that do not play any role.

that $\lambda j.\ F(\tilde{\mathbf{s}}, \mathbf{z}, \mathbf{a}, \bar{1}, j) = \mathsf{store}(\mathbf{a}, \kappa(\tilde{\mathbf{s}}), \mathbf{t}(\tilde{\mathbf{s}}, \mathbf{z}, \mathbf{a}(\kappa(\tilde{\mathbf{s}})), \mathbf{a}[\mathbf{z}]))$ holds (the latter because for every h, $\iota_h(\tilde{\mathbf{s}}, j) = 0 \wedge j = \kappa_h(\mathbf{u}^*(\mathbf{s}, \iota_h(\tilde{\mathbf{s}}, j))$ is equivalent to $j = \kappa_h(\mathbf{u}^*(\tilde{\mathbf{s}}, 0)) = \kappa_h(\tilde{\mathbf{s}})$ by (5.8)).

For the induction step, we suppose that Lemma 5.2.1 holds for n and show it for $n+1$. As a preliminary remark, notice that from (5.9), we get not only $\mathbf{z}' = \mathbf{z}$, but also $\mathbf{a}'[\mathbf{z}'] = \mathbf{a}[\mathbf{z}]$, because of (v) of Definition 5.2.4. As a consequence, after n iterations of τ, the values $\mathbf{z}, \mathbf{a}[\mathbf{z}]$ are left unchanged; thus, for notation simplicity, we will not display anymore below the dependence of $\phi_L, \tilde{\mathbf{t}}$ on $\mathbf{z}, \mathbf{a}[\mathbf{z}]$. We need to show that $\tau \circ \tau^n$ matches the required shape (5.10)-(5.11) with $n+1$ instead of n. After unraveling the definitions, this splits into three sub-claims, concerning the update of the $\mathbf{s}$, the guard and the update of the $\mathbf{a}$, respectively:

(i) the equality $\mathbf{u}(\mathbf{u}^*(\tilde{\mathbf{s}}, \bar{n})) = \mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n+1})$ is valid;

(ii)

$$\bigwedge_{0 \le k < n} \tilde{\phi}_L(\mathbf{u}^*(\tilde{\mathbf{s}}, \bar{k}), \mathbf{a}(\kappa(\mathbf{u}^*(\tilde{\mathbf{s}}, \bar{k})))) \wedge \tilde{\phi}_L(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n}), \lambda j.\ F(\mathbf{s}, \mathbf{a}, \bar{n}, j)(\kappa(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n}))))$$

is equivalent to

$$\bigwedge_{0 \le k < n+1} \tilde{\phi}_L(\mathbf{u}^*(\mathbf{s}, \bar{k}), \mathbf{a}(\kappa(\mathbf{u}^*(\mathbf{s}, \bar{k}))))$$

(iii)

$$\mathsf{store}(\lambda j.\ F(\mathbf{s}, \mathbf{a}, \bar{n}, j), \kappa(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n})), \tilde{\mathbf{t}}(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n}), \lambda j.\ F(\mathbf{s}, \mathbf{a}, \bar{n}, j)(\kappa(\mathbf{u}^*(\tilde{\mathbf{s}}, \bar{n})))))$$

is the same function as

$$\lambda j.\ F(\mathbf{s}, \mathbf{a}, \overline{n+1}, j)$$

Statement (i) is trivial, because $\mathbf{u}(\mathbf{u}^*(\tilde{\mathbf{s}}, \bar{n})) = \mathbf{u}(\mathbf{u}^n(\tilde{\mathbf{s}})) = \mathbf{u}^{n+1}(\tilde{\mathbf{s}}) = \mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n+1})$ holds by (5.6).

To show (ii), it is sufficient to check that

$$\mathbf{a}(\kappa(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n}))) \ = \ \lambda j.\ F(\mathbf{s}, \mathbf{a}, \bar{n}, j)(\kappa(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n}))) \tag{5.12}$$

is true. In turn, this follows from (5.11) and the validity of the following

implications (varying $h = 1, \ldots, r$)

$$\iota_h(\tilde{\mathbf{s}}, j) \neq \bar{n} \;\rightarrow\; j \neq \kappa_h(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n})) \tag{5.13}$$

(in fact, a_h and F_h can possibly differ only for the j satisfying $0 \leq \iota_h(\tilde{\mathbf{s}}, j) < \bar{n}$, i.e. in particular for the j such that $\iota_h(\tilde{\mathbf{s}}, j) \neq n$). To see why (5.13) is valid, notice that in view of (5.7), what (5.13) says is that we cannot have simultaneously both $\iota_h(\tilde{\mathbf{s}}, j) = \overline{n}$ and $\iota_h(\tilde{\mathbf{s}}, j) = \overline{m}$, for some $m \neq n$: indeed it is so by the definition of a function.

It remains to prove (iii); in view of (5.12) just shown, we need to check that $\mathsf{store}(\lambda j.\, F(\mathbf{s}, \mathbf{a}, \bar{n}, j), \kappa(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n})), \tilde{\mathbf{t}}(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n}), \mathbf{a}(\kappa(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n})))))$ is the same as $\lambda j.\; F(\mathbf{s}, \mathbf{a}, \overline{n+1}, j)$. For every $h = 1, \ldots, r$, this is split into three cases, corresponding to the validity check for the three implications:

$$\begin{aligned}
&i_h(\tilde{\mathbf{s}}, j) < \bar{n} \rightarrow \mathsf{store}(\lambda j.\, F_h(\mathbf{s}, \mathbf{a}, \bar{n}, j), \kappa_h(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n})), \tilde{t}_h)(j) = F_h(\mathbf{s}, \mathbf{a}, \overline{n+1}, j)\\
&i_h(\tilde{\mathbf{s}}, j) = \overline{n} \rightarrow \mathsf{store}(\lambda j.\, F_h(\mathbf{s}, \mathbf{a}, \bar{n}, j), \kappa_h(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n})), \tilde{t}_h)(j) = F_h(\mathbf{s}, \mathbf{a}, \overline{n+1}, j)\\
&i_h(\tilde{\mathbf{s}}, j) > \overline{n} \rightarrow \mathsf{store}(\lambda j.\, F_h(\mathbf{s}, \mathbf{a}, \bar{n}, j), \kappa_h(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n})), \tilde{t}_h)(j) = F_h(\mathbf{s}, \mathbf{a}, \overline{n+1}, j)
\end{aligned}$$

where we wrote simply $\tilde{t}_h$ instead of $\tilde{t}_h(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n}), \mathbf{a}(\kappa(\mathbf{u}^*(\tilde{\mathbf{s}}, \overline{n}))))$. However, keeping in mind (5.13) and (5.8), the three implications can be rewritten as follows (the second one is split into two subcases)

$$\begin{aligned}
&i_h(\tilde{\mathbf{s}}, j) < \bar{n} \rightarrow\; F_h(\mathbf{s}, \mathbf{a}, \bar{n}, j) = F_h(\mathbf{s}, \mathbf{a}, \overline{n+1}, j)\\
&i_h(\tilde{\mathbf{s}}, j) = \overline{n} \wedge j = \kappa_h(\mathbf{u}^*(\mathbf{s}, \iota_h(\tilde{\mathbf{s}}, j))) \rightarrow \tilde{t}_h = F_h(\mathbf{s}, \mathbf{a}, \overline{n+1}, j)\\
&i_h(\tilde{\mathbf{s}}, j) = \overline{n} \wedge j \neq \kappa_h(\mathbf{u}^*(\mathbf{s}, \iota_h(\tilde{\mathbf{s}}, j))) \rightarrow\; F_h(\mathbf{s}, \mathbf{a}, \bar{n}, j) = F_h(\mathbf{s}, \mathbf{a}, \overline{n+1}, j)\\
&i_h(\tilde{\mathbf{s}}, j) > \overline{n} \rightarrow\; F_h(\mathbf{s}, \mathbf{a}, \bar{n}, j) = F_h(\mathbf{s}, \mathbf{a}, \overline{n+1}, j)
\end{aligned}$$

The above four implications all hold by the definitions (5.11) of the F_h. □

Theorem 5.2.2. *If τ is a local ground assignment, then τ^+ is a Σ_2^0-assignment.*

Proof. As a preliminary observation, we notice that the bi-implications of the kind

$$\bigvee_{n \geq 0} \psi(\underline{x}, \bar{n}) \;\leftrightarrow\; \exists y\, (y \geq 0 \wedge \psi(\underline{x}, y)) \tag{5.14}$$

are valid because we interpret our formulæ in the *standard* structure of natural numbers (enriched with extra free symbols).

As a second preliminary observation, we notice that (5.7) can be equiva-

lently re-written in the form of a bi-implication as:

$$z = \kappa(\mathbf{u}^*(\underline{x}, y)) \quad \leftrightarrow \quad [\, y = \iota(\underline{x}, z) \ \wedge \ z = \kappa(\mathbf{u}^*(\underline{x}, \iota(\underline{x}, z)))\,] \tag{5.15}$$

(to see why (5.15) is equivalent to (5.7) it is sufficient to apply the logical laws of pure identity).

Let us fix a local ground assignment of the form (5.9); let $\mathbf{a}[\mathbf{z}]$ indicate the $r \cdot |\mathbf{z}|$-tuple of terms $\{a_i(z_j)\}_{1 \leq i \leq r, 1 \leq j \leq |\mathbf{z}|}$; since ϕ_L and $\mathbf{t}$ are purely arithmetical over $\{\tilde{\mathbf{s}}, \mathbf{z}, \mathbf{a}(\kappa(\tilde{\mathbf{s}})), \mathbf{a}[\mathbf{z}]\}$, we have that they can be written as $\tilde{\phi}_L(\tilde{\mathbf{s}}, \mathbf{z}, \mathbf{a}(\kappa(\tilde{\mathbf{s}})), \mathbf{a}[\mathbf{z}])$, $\tilde{\mathbf{t}}(\tilde{\mathbf{s}}, \mathbf{z}, \mathbf{a}(\kappa(\tilde{\mathbf{s}})), \mathbf{a}[\mathbf{z}])$, respectively, where $\tilde{\phi}_L, \tilde{\mathbf{t}}$ do not contain occurrences of the free function and constant symbols $\mathbf{a}, \mathbf{s}$.

As a consequence of the Lemma 5.2.1, since the formula

$$\bigwedge_{0 \leq k < n} \tilde{\phi}_L(\mathbf{u}^*(\tilde{\mathbf{s}}, \bar{k}), \mathbf{z}, \mathbf{a}(\kappa(\mathbf{u}^*(\tilde{\mathbf{s}}, \bar{k}))), \mathbf{a}[\mathbf{z}])$$

is equivalent to

$$\forall z \ (0 \leq z < \bar{n} \rightarrow \tilde{\phi}_L(\mathbf{u}^*(\tilde{\mathbf{s}}, z), \mathbf{z}, \mathbf{a}(\kappa(\mathbf{u}^*(\mathbf{s}, z))), \mathbf{a}[\mathbf{z}]))$$

we can use (5.14) to express τ^+ as

$$\exists y > 0 \left(\begin{array}{l} \forall z \ (0 \leq z < y \rightarrow \tilde{\phi}_L(\mathbf{u}^*(\tilde{\mathbf{s}}, z), \mathbf{z}, \mathbf{a}(\kappa(\mathbf{u}^*(\mathbf{s}, z))), \mathbf{a}[\mathbf{z}])) \wedge \mathbf{z}' = \mathbf{z} \wedge \\ \quad \wedge \ pc = l \ \wedge \ pc' = l \ \wedge \ \tilde{\mathbf{s}}' = \mathbf{u}^*(\tilde{\mathbf{s}}, y) \ \wedge \ \mathbf{a}' = \lambda j.\ F(\mathbf{s}, \mathbf{a}, y, j) \end{array} \right) \tag{5.16}$$

The latter shows that τ^+ is a Σ_2^0-assignment, as desired. □

Example 5.2.6. Consider again our Reverse running example and its transition τ_1, i.e.,

$$pc = l_2 \wedge c \neq N + 1 \wedge pc' = l_2 \wedge c' = c + 1 \wedge I' = I \wedge O' = \mathsf{store}(O, c, I(N - c)) \ .$$

Notice that the variable pc is left unchanged in this transition (this is essential, otherwise the acceleration gives an inconsistent transition that can never fire). If we accelerate it, we get the Σ_2^0-assignment

$$\exists y > 0 \left(\begin{array}{l} pc = 2 \wedge \forall j \ (c \leq j < c + y \rightarrow j \neq N + 1) \wedge c' = c + y \ \wedge \\ O' = \lambda j \ (\mathsf{if} \ c \leq j < c + y \ \mathsf{then} \ I(N - j) \ \mathsf{else} \ O(j)) \end{array} \right) \tag{5.17}$$

Paraphrasing this formula, it says: take an integer y greater than 0 (which represents the arbitrary number of applications of τ_1). If all the positions from

c to $c + y - 1$ are within the array bounds, i.e., below the upper-bound N, we can copy *at once* the content of these positions of I into O in the revert order.

5.2.3 Sub-fragments of acceleratable assignments

From an implementation point of view, the effectiveness of the acceleration procedure showed in this chapter depends on the availability of a repository of iterators and selectors.

In many applications it is sufficient to consider a subclass of local ground assignments. This simplifies things and allows to establish classes of assignments that can be matched more easily and accelerated without requiring big repositories of iterators and selectors.

For the first subclass we consider, $\mathbf{s}$ is a single counter $\mathbf{s}$ that is incremented by one (otherwise said, the iterator is $x_1 + 1$) and the selector assignment is trivial, namely it is just x_1. We call these local ground assignments *simple*.

Definition 5.2.5 (Simple local ground assignments)**.** *Thus, a simple local ground assignment has the form*

$$pc = l \ \wedge \ \phi_L(\mathbf{s}, \mathbf{a}) \ \wedge \ pc' = l \wedge \mathbf{s}' = \mathbf{s} + 1 \ \wedge \ \mathbf{a}' = \mathsf{store}(\mathbf{a}, \mathbf{s}, \mathbf{t}(\mathbf{s}, \mathbf{a})) \quad (5.18)$$

where the first occurrence of $\mathbf{s}$ *in* $\mathsf{store}(\mathbf{a}, \mathbf{s}, \mathbf{t}(\mathbf{s}, \mathbf{a}))$ *stands in fact for an* s*-tuple of terms all identical to* $\mathbf{s}$*, and where* $\phi_L, \mathbf{t}$ *are purely arithmetical over the terms* $\mathbf{s}$*,* $a_1[\mathbf{s}], \ldots, a_r[\mathbf{s}]$*.*

Proposition 5.2.1. *The accelerated transition computed in the proof of Theorem 5.2.2 for* (5.18) *can be rewritten as follows:*

$$\exists k \left(\begin{array}{c} k > 0 \ \wedge \ pc = l \ \wedge \ \forall j \ (\mathbf{s} \leq j < \mathbf{s} + k \rightarrow \phi_L(j, \mathbf{a})) \ \wedge \ pc' = l \ \wedge \\ \wedge \ \mathbf{s}' = \mathbf{s} + k \ \wedge \ \mathbf{a}' = \lambda j. \ (\texttt{if } \mathbf{s} \leq j < \mathbf{s} + k \texttt{ then } \mathbf{t}(j, \mathbf{a}) \texttt{ else } \mathbf{a}[j]) \end{array} \right) \quad (5.19)$$

Programs with nested loops (e.g., sorting procedures) might need an extension of simple local ground assignments. This happens when an array is scanned by a couple of counters, one of which is kept fixed (this is the case of inner loops of sorting algorithms). To cope with these more complicated cases, we introduce a larger class of assignments, still local, hence covered by Theorem 5.2.2).

Definition 5.2.6 (Simple+ local ground assignments)**.** *We call* simple+ *the*

local ground assignments of the form

$$pc = l \ \wedge \ \phi_L(\mathbf{s}, \mathbf{z}, \mathbf{a}) \ \wedge \ pc' = l \wedge \mathbf{s}' = \mathbf{s} \pm 1 \ \wedge \ \mathbf{z}' = \mathbf{z} \ \wedge \ \mathbf{a}' = wr(\mathbf{a}, \mathbf{s}, \mathbf{t}(\mathbf{s}, \mathbf{z}, \mathbf{a})) \tag{5.20}$$

where

(i) $\mathbf{z} = z_1, \ldots, z_q$ *is a tuple of integer constants,*

(ii) the first occurrence of $\mathbf{s}$ *in* $wr(\mathbf{a}, \mathbf{s}, \mathbf{t}(\mathbf{s}, \mathbf{z}, \mathbf{a}))$ *stands for a tuple of terms all identical to* $\mathbf{s}$*,*

(iii) the guard ϕ_L *contains the conjuncts* $\mathbf{s} \neq z_i$ *(*$1 \leq i \leq q$*), and*

(iv) $\phi_L, \mathbf{t}$ *are purely arithmetical over* $\mathbf{s}, \mathbf{z}, a_1[\mathbf{s}], \ldots a_r[\mathbf{s}], a_1[z_1], \ldots, a_r[z_q]$.

Basically, simple+ local ground assignments differ from plain simple ones just because there are some 'idle' indices $\mathbf{z}$; in addition, the counter $\mathbf{s}$ can also be decremented.

Proposition 5.2.2. *The accelerated transition for* (5.20) *computed by Theorem 5.2.2 can be re-written as follows (we write* $j \in [\mathbf{s}, \mathbf{s} \pm k]$ *for* $\mathbf{s} \leq j \leq \mathbf{s} + k$ *or* $\mathbf{s} - k \leq j \leq \mathbf{s}$*, depending on whether we have increment or decrement in* (5.20)*):*

$$\exists k \left(\begin{array}{c} k > 0 \ \wedge \ pc = l \ \wedge \ \forall j \ (j \in [\mathbf{s}, \mathbf{s} \pm k] \rightarrow \phi_L(j, \mathbf{z}, \mathbf{a})) \ \wedge \ pc' = l \ \wedge \ \mathbf{z}' = \mathbf{z} \ \wedge \\ \wedge \ \mathbf{s}' = \mathbf{s} \pm k \ \wedge \ \mathbf{a}' = \lambda j. \ (\texttt{if} \ j \in [\mathbf{s}, \mathbf{s} \pm k] \ \texttt{then} \ \mathbf{t}(j, \mathbf{z}, \mathbf{a}) \ \texttt{else} \ \mathbf{a}[j]) \end{array} \right) \tag{5.21}$$

Other classes of local ground assignments admitting definable acceleration that are particularly useful in practice will be introduced in chapter 6.

5.3 Acceleration-based backward reachability and monotonic abstraction

The particular shape of accelerated transitions, i.e., Σ^0_2-sentences, invalidates the direct application of acceleration in the BReach procedure. The generation of Σ^0_2-formulæ is problematic since they might invalidate both the safety and the fixpoint tests of the BReach procedure.

Example 5.3.1. Let us again consider the formalization of the Reverse procedure given in Example 5.1.1. Suppose that in a preprocessing step we add the

accelerated transition τ_1^+ given by (5.17) to the transitions we already have. This causes the generation of a new leaf labeled with

$$\exists n, z_1, z_2 \left(\begin{array}{l} n > 0 \wedge pc = l_1 \ \wedge \ \forall j.(c \leq j < c + n \rightarrow j \neq N + 1) \wedge \\ c + n = N + 1 \wedge z_1 \geq 0 \wedge z_2 \geq 0 \wedge z_1 + z_2 = N \ \wedge \\ I(z_1) \neq \lambda j.(\mathsf{if}\ c \leq j < c + n\ \mathsf{then}\ I(N - j)\ \mathsf{else}\ O(j))(z_2) \end{array} \right)$$

that, simplified, can be rewritten as

$$\exists n, z_1, z_2 \left(\begin{array}{l} n > 0 \wedge pc = l_1 \ \wedge \ \forall j.(c \leq j < c + n \rightarrow j \neq N + 1) \wedge \\ c + n = N + 1 \wedge z_1 \geq 0 \wedge z_2 \geq 0 \wedge z_1 + z_2 = N \ \wedge \\ ((c > z_2 \vee z_2 \geq c + n) \wedge I(z_1) \neq O(z_2)) \end{array} \right)$$

The solution we propose is to over-approximate such sentences by adopting a selective instantiation schema, known in literature as *monotonic abstraction*.

5.3.1 Monotonic Abstraction

Monotonic abstraction is a technique introduced by P. A. Abdulla et al. in a series of papers (e.g., [Abdulla et al., 2007a, Abdulla et al., 2007b, Abdulla et al., 2008b, Abdulla, 2010]), originally applied in the context of verification of distributed systems. In this section we will briefly describe its origin and its application in parameterized model-checking. Then we will show how it is possible to import it in our framework.

In the seminal paper [Abdulla et al., 1996], the authors introduce the notion of infinite-state systems which are monotonic w.r.t. a well quasi-ordering on the set of configurations. That is, the set of configurations of a system is endowed with a well-quasi ordering $\preceq$, i.e., a reflexive, transitive binary relation that neither contains infinite strictly decreasing sequences nor infinite sequences of pairwise incomparable elements, such that $\preceq$ is a simulation with respect to the transition relation, or, in other words, the transition relation is monotonic with respect to $\preceq$. This definition is suitable for checking, via a backward reachability analysis, the safety of parameterized systems. Set of unsafe states are represented by an upward closed set K such that, for any state s', if $s \in K$ and $s \preceq s'$, then $s' \in K$. Moreover, monotonicity of the transition relation with respect to $\preceq$ implies that the pre-image of an upward closed set is still an upward closed set. Finally, since $\preceq$ is a well-quasi ordering, upward closed sets can be *finitely* represented by their finitely many minimal elements.

The notion of array-based transition system $\mathcal{S}_\mathcal{T}$ reinterprets this idea in a

declarative framework [Ghilardi and Ranise, 2010a]. The nature of $\mathcal{T}$ depends on the application domain where we are working: in parameterized distributed systems, usually, there is no arithmetic on index sort (processes are just ordered) and $\mathcal{T}$ is quite simple. In this context, if we view the system variables $\mathbf{v}$ as fresh constants, configurations can be identified with finitely generated models of $\mathcal{T}$ (with generators having all `INDEX` sort), ordering among configurations is model-theoretic embeddability and upward closed sets are characterizable via definability with Σ^0_1-sentences. As stated before in Proposition 5.1.1, if the unsafe formula is represented by a Σ^0_1-sentence and transition relations are all Σ^0_1-assignments, BReach can generate only Σ^0_1-sentences.

In such a context, a Σ^0_1-assignment are used to represent transitions like "if there are two processes p_1 and p_2 in location Waiting and p_1 has an id smaller than p_2, then p_1 can enter the Critical section". This transition is represented by the Σ^0_1-assignment

$$\exists p_1 \exists p_2 . (l[p_1] = \mathsf{W} \wedge l[p_2] = \mathsf{W} \wedge p_1 < p_2 \wedge l' = \lambda j.(\mathsf{if}\ (j = p_1)\ \mathsf{then}\ \mathsf{C}\ \mathsf{else}\ l[j]))$$

Σ^0_1-assignments are not sufficient for representing transitions where a process has to check the status of *all* other processes of the system, though. In this case, Σ^0_2-assignments are needed. Consider, for example, a protocol where a process in the Waiting location enters the Critical section only if its id is lower than the id of all the other processes in the Waiting section. In this case, the transition will look like

$$\exists p_1 . \begin{pmatrix} l[p_1] = \mathsf{W} \wedge \forall p_j . (l[p_j] = \mathsf{W} \rightarrow p_1 < p_j) \wedge \\ l' = \lambda j.(\mathsf{if}\ (j = p_1)\ \mathsf{then}\ \mathsf{C}\ \mathsf{else}\ l[j]) \end{pmatrix}$$

The preimage along Σ^0_2-assignments do not yield existential formulæ. This destroys the entire framework, and here is when monotonic abstraction comes into play. The Σ^0_2-sentences obtained as preimages of Σ^0_2-assignments are over-approximated with their monotonic abstraction. In the setting of distributed systems, applying monotonic abstraction techniques amounts to adopting the "stopping failures" computational model [Lynch, 1996]. We assume that processes can crash at any instant of time and that crashed processes do not take part anymore to the protocol. In this setting, a Σ^0_2-assignment can always fire, provided the processes violating the universal guard $\forall \underline{j} . \psi_U(\mathbf{s}, \mathbf{a}, \underline{k}, \underline{j})$ crash. Notably, this transformation can be interpreted as a modification of the underlying computational model (we are adopting the "stopping failures" paradigm) or more simply just as a kind of abstraction. One should be aware that the

modified system has more runs, so safety of the modified system implies safety of the original one but not vice versa. However, the point is that in the context of array-based transition system, this monotonic abstraction modification can be performed at the syntactic level: by using quantifier relativizations and by adding a "crash" case to the update function G, it is possible to transform a Σ_0^2-assignment into a Σ_1^0-assignment[4].

Monotonic abstraction has been applied in different application domains (see e.g. [Abdulla et al., 2008a, Abdulla et al., 2009]). Its reformulation has been exploited within within the declarative context of array-based transition systems in order to apply it to the verification of reliable broadcast algorithms in a fault-tolerant environment [Chandra and Toueg, 1990, Alberti, 2010].

What allows to import monotonic abstraction into the context of this thesis, i.e., verification of sequential programs with arrays, is the following: the declarative reformulation clearly shows that monotonic abstraction can be viewed operationally as a purely symbolic manipulation applying quantifier instantiation in order to overapproximate sets of states represented via Σ_2^0-sentences.

Definition 5.3.1 (Monotonic abstraction)**.** *Let*

$$\psi \equiv \exists \underline{i}\, \forall \underline{j}.\ \phi(\underline{i}, \underline{j}, \mathbf{a}, \mathbf{s}, pc)$$

be a Σ_2^0-sentence and let $\mathcal{I}$ be a finite set of terms of the form $t(\underline{i}, \mathbf{v})$. The monotonic $\mathcal{I}$-approximation of ψ *is the Σ_1^0-sentence*

$$\exists \underline{i} \bigwedge_{\sigma : \underline{j} \to \mathcal{I}} \phi(\underline{i}, \underline{j}\sigma / \underline{j}, \mathbf{a}, \mathbf{s}, pc) \tag{5.22}$$

(here $\underline{j}\sigma$ is the tuple of terms $\sigma(j_1), \ldots, \sigma(j_n)$, where $\underline{j} = j_1, \ldots, j_n$,).

By Definition 5.3.1, universally quantified variables are *eliminated through instantiation*; the larger the set $\mathcal{I}$ is, the better approximation we get. In practice, the natural choices for $\mathcal{I}$ are $\underline{i}$ or the set of terms of the kind $t(\underline{i}, \mathbf{v})$ occurring in ψ As a result of replacing Σ_2^0-sentences by their monotonic approximation, spurious unsafe traces might occur. However, those can be disregarded if accelerated transitions contribute to their generation. This is because if $\mathcal{S}_\mathcal{T}$ is unsafe, its unsafety can be discovered without considering accelerated transitions.

[4]For more information, the interested reader is pointed to [Alberti et al., 2012d].

5.3.2 An acceleration-based backward reachability procedure

To integrate monotonic abstraction, the BReach procedure is modified as follows. In a preprocessing step, we analyze the input array-based transition system, produce accelerated transitions following the procedure described in section 5.2 and *add* these transitions to $\mathcal{S}_{\mathcal{T}}$.

The procedure BReach is therefore modified in the ABReach procedure (Figure 5.3). It is quite straightforward to see that Proposition 5.1.1 applies to ABReach as well. Notice that, contrarily to what happens in other acceleration-based approaches for integer variables, e.g., [Bozga et al., 2014], we *do not substitute* the "acceleratable" transitions with their accelerated counterpart, but we *add* the accelerations to the set T. This is because the pre-images of Σ_2^0-assignment will be over-approximated with their monotonic abstraction. This process can cause spurious counterexamples. In case of unsafety, the algorithm checks if the counterexample is spurious. If it contains an accelerated transition τ^+, the subtree having as a root the vertex labeled with the monotonic abstraction of the pre-image obtained along τ^+ is removed. This implies that spurious traces containing approximated accelerated transitions cannot be produced again and again: when the sub-tree D from the target node v of τ^+ is removed by Check', the node v is not a leaf (the arcs labeled by the transitions τ are still there), hence it cannot be expanded anymore according to the Expansion instruction.

Example 5.3.2. We resume from Example 5.3.1. Consider the label of v_2^+, i.e., the vertex of the labeled unwinding having as a preimage $Pre(\tau_1^+, M_V(v_1))$, represented by the formula

$$\exists n > 0\, \exists z_1, z_2 \begin{pmatrix} pc = l_2 \ \wedge\ \forall j.(c \leq j < c + n \rightarrow j \neq N + 1) \wedge \\ c + n = N + 1 \wedge z_1 \geq 0 \wedge z_2 \geq 0 \wedge z_1 + z_2 = N \wedge \\ I(z_1) \neq \lambda j.(\mathsf{if}\ c \leq j < c + n\ \mathsf{then}\ I(N - j)\ \mathsf{else}\ O(j))(z_2) \end{pmatrix}$$

We approximate using the set of terms $\mathcal{I} = \{z_1, z_2, n\}$. After simplifications we get

$$\exists z_1, z_2.(pc = l_1 \wedge c \leq N \wedge z_1 \geq 0 \wedge z_2 \geq 0 \wedge z_1 + z_2 = N \wedge O(z_2) \neq I(z_1) \wedge c > z_2)$$

Generating this formula is enough to allow the convergence of ABReach.

ABReach

Initialization: a single node tree labeled by $pc = l_E$ and is marked 'unchecked'.

Check': pick an unchecked leaf L labeled by a formula K. If K is a Σ_2^0-sentence, choose a suitable $\mathcal{I}$ and replace K by its monotonic abstraction K'. If $K' \wedge pc = l_I$ is inconsistent, mark L as 'covered' or 'checked' according to the outcome of the fixpoint check, as was done in the original Check. If $K' \wedge pc = l_I$ is satisfiable, analyze the path from the root to L. If no accelerated transition τ^+ is found in it return unsafe, otherwise remove the sub-tree D from the target of τ^+ to the leaves. Each node N covered by a node in D will be flagged as 'unchecked' (to make it eligible in future for the Expansion instruction).

Expansion: pick a checked leaf L labeled with K. For each transition $\tau_i \in T$, add a new leaf below L labeled with $Pre(\tau_i, L)$ and marked as 'unchecked'. The arc between L and the new leaf is labeled with τ_i.

Safety Exit: if all leaves are covered, exit and return safe.

Figure 5.3. The ABReach backward reachability procedure.

5.4 Experimental evaluation

In this section we show that ABReach, i.e., the backward reachability procedure enhanced to handled Σ_2^0-sentences arising as pre-images along accelerated transitions, is able to check the safety of a set of problems with arrays.

For our experimental evaluation we will exploit the mcmt model-checker [Alberti et al., 2014d], offering an implementation of ABReach procedure.

Notably, acceleration requires a preprocessing of the input program to compactly represent the transition relation. That is, loops have to be represented by a single transition. This program representation might be far from the usual internal representation of the input code generated by standard compilers infrastructures (see, e.g., [Aho et al., 2007]). In our case, as we will discuss in chapter 8, it is beneficial to represent the program as a *cutpoint graph* [Gurfinkel et al., 2011]. We are not going into the details of such representation now: the discussion is delayed to section 8.1.1. Here we only observe that the benchmarks we are using for our experimental evaluation have been preprocessed by Booster, the tool we will present in chapter 8. Without this preprocessing phase, mcmt is not able to converge on any example.

In this section we want to compare the solution presented in chapters 3 and 4 with the acceleration-based approach of this chapter. We take the two benchmark suites used to evaluate SAFARI. The running times, with timeout set to 1 hour, are reported in Table 5.1 and Table 5.2.

PB. (L,N,Q)	STATUS	SAFARI	MCMT
D01 (2,0,0)	safe	0.38	0.06
D02 (2,0,0)	safe	0.28	0.06
D03 (2,0,0)	safe	0.52	0.05
D04 (2,0,0)	unsafe	0.28	0.04
D06 (2,0,0)	unsafe	0.78	0.03
D08 (2,0,0)	safe	0.50	0.06
D09 (2,0,0)	safe	0.40	0.06
D11 (2,0,0)	unsafe	1.02	0.03
D13 (2,0,0)	unsafe	0.45	0.08
D14 (4,0,0)	safe	1.06	0.40
D15 (4,0,0)	unsafe	1.56	0.26
D16 (5,0,0)	safe	1.10	0.52
D17 (2,0,0)	safe	0.68	x
D20 (2,0,0)	safe	0.47	0.08
QD01 (1,0,1)	safe	0.39	0.03
QD02 (1,0,1)	safe	0.35	0.06
QD03 (1,0,1)	safe	0.38	0.04
QD04 (1,0,1)	unsafe	0.37	0.03
QD08 (1,0,1)	safe	0.21	0.03
QD09 (1,0,1)	safe	0.44	0.42
QD11 (1,0,1)	unsafe	0.58	0.03
QD13 (2,0,2)	unsafe	0.35	0.10
QD14 (3,0,1)	safe	0.64	0.16
QD15 (3,0,1)	unsafe	0.94	0.10
QD16 (4,0,1)	safe	1.14	0.22
QD20 (1,0,1)	safe	0.28	0.04

Table 5.1. Experiments on SUITE 1: running time for SAFARI and MCMT. SAFARI has been executed with both Term Abstraction and Counterexample Minimization enabled. A 'x' indicates that the tool was not able to converge in the given time out of 1 hour.

The two tables confirm the starting conjecture, i.e., on one side acceleration gives better results (in matter of running time) than abstraction, thanks to its precision. On the other side, abstraction achieve a complete coverage of the

	SAFE		UNSAFE	
PB. (L,N,Q)	SAFARI	MCMT	SAFARI	MCMT
binarySort	3.95	x	6.53	2.00
bubbleSort	1.04	x	8.26	2.82
comp	0.32	0.04	0.40	0.06
compM	0.38	0.06	0.48	0.03
copy	0.28	0.03	0.34	0.03
copy2	0.33	0.06	0.45	0.05
copy3	0.39	0.08	0.57	0.15
copy4	0.45	0.14	0.78	0.74
copy5	0.50	0.19	0.98	2.63
copy6	0.57	0.21	1.22	8.31
copy7	0.64	0.21	1.51	20.31
copy8	0.67	0.28	1.76	48.97
copy9	0.72	0.46	2.17	104.22
copy10	0.80	0.52	2.57	245.78
find	0.28	0.03	0.36	0.05
findTest	0.41	0.03	0.85	0.13
heapArr	0.34	x	0.51	0.05
init	0.29	0.03	0.31	0.03
initTest	0.35	0.10	0.42	0.12
maxInArr	0.29	0.04	0.38	0.06
minInArr	0.29	0.04	0.39	0.06
nonDisj	0.55	1.39	0.76	2.50
partition	2.24	x	0.61	0.06
running	0.28	0.50	0.46	0.13
vararg	0.19	0.03	0.35	0.04

Table 5.2. Experiments on SUITE 2: running time for SAFARI and MCMT. SAFARI has been executed with both Term Abstraction and Counterexample Minimization enabled. A 'x' indicates that the tool was not able to converge in the given time out of 1 hour.

examples while acceleration fails on some examples, e.g., those with a more complex data-flow structure like sorting procedures.

Given that we are working with a declarative framework, we can easily combine the two techniques. Indeed, this idea is at the core of the BOOSTER tool, presented in chapter 8. There we shall show that acceleration and abstraction can be gainfully combined, in such a way that the resulting framework overcomes their individual limitations and allows to achieve very good results in

practice.

5.5 Related work

Acceleration has been widely and successfully applied to systems modeled via integer state variables: indeed, transitive closure can be computed precisely (it is definable within Presburger arithmetic) for relations that can be formalized as difference bounds constraints [Comon and Jurski, 1998,Bozga et al., 2009c], octagons [Bozga et al., 2009a] and finite monoid affine transformations [Finkel and Leroux, 2002] (paper [Bozga et al., 2010] presents a general approach covering all these domains). Recently, acceleration for relations over Presburger arithmetic has been plugged into abstraction/refinement loop for verifying integer programs [Hojjat et al., 2012]. In contrast, our work, to best of our knowledge, for the first time extends acceleration and its integration with abstraction/refinement to verification of array-based programs. A first promising technique allowing acceleration of relations involving arrays of integers is presented in [Bozga et al., 2009b] via counter automata encoding. This solution seems to be unable to handle properties of common interest with more than one quantified variable (e.g., "sortedness") and is limited to programs without nested loops.

Acceleration has also been applied proficiently in the analysis of real time systems (e.g., [Hendriks and Larsen, 2002,Behrmann et al., 2002]), to represent in one transition the iterated execution of cyclic actions (e.g., polling-based systems) and address the fragmentation problem.

As another related work to this area, it is worth to mention Cook's completeness proof which reduces safety to an arithmetic encoding [Cook, 1978].

5.6 Summary

In this chapter we addressed the problems of (i) identifying a class of relations over arrays admitting definable acceleration modulo the theory of linear arithmetic over the integers enriched with free function symbols and (ii) studying how to exploit acceleration into practice in the context of the verification of programs with arrays.

We presented the divergence problem suffered by precise backward reachability analysis (section 5.1), detected a class of relations over arrays admitting definable accelerations and show how to compute such acceleration in practice (section 5.2). Definability of accelerations comes at the price of introducing

nested quantifiers that might prevent the practical exploitation of acceleration. This might prevent practical advantages of acceleration for arrays. Our solution for this problem is the introduction of a quantifier-instantiation procedure called *monotonic abstraction* (section 5.3), coupled with a suitable refinement procedure.

We experimentally evaluated a prototype implementing the acceleration procedure along with the aforementioned monotonic abstraction and refinement techniques (section 5.4). The results show that acceleration constitutes an alternative to abstraction on some benchmarks, leading to the conjecture (that will be confirmed with the combined framework of chapter 8) that abstraction and acceleration are orthogonal techniques that can be gainfully combined together.

5.6.1 Related publications

The results reported in this chapter have been published in the following paper:

- F. Alberti, S. Ghilardi, and N. Sharygina. Definability of accelerated relations in a theory of arrays and its applications. In P. Fontaine, C. Ringeissen, and R. A. Schmidt, editors, *Frontiers of Combining Systems - 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings*, volume 8152 of *Lecture Notes in Computer Science*, pages 23–39. Springer, 2013.

Chapter 6

Decision procedures for Flat Array Properties

This chapter presents new decidable quantified fragments of the theories of arrays. A central notion in formal verification is the one of *proof obligation*. Frameworks for the formal verification of systems, like those presented in Chapters 3, 4 and 5, are usually structured according to a client-server architecture. The client side implements algorithms and procedures in charge of analyzing the input system, and the server is usually represented by a theorem prover dealing with the queries generated by the client encoded in logical terms, the so-called proof obligations. Proof obligations are, therefore, (first-order) formulæ whose satisfiability or unsatisfiability drives the verification process. As we already discussed in section 2.2, to achieve our final goal, i.e., the formal verification of programs with arrays, quantification is required over the indexes of the arrays in order to express significant properties like "the array has been initialized to 0" or "there exist two different positions of the array containing an element c", for example.

In this chapter we focus our attention on the fragment

$$\exists \mathbf{c}\, \forall \mathbf{i}\ \psi(\ \mathbf{c}\ ,\ \mathbf{i}\ ,\ \mathbf{a}(t)\ ) \tag{2.2}$$

of the theories of arrays $\mathtt{ARR}^1(\mathcal{T})$ or $\mathtt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$.

We recall that from a logical point of view, array variables are interpreted as functions; adding free function symbols to a theory $\mathcal{T}$ or $\mathcal{T}_I \cup \mathcal{T}_E$ (with the goal to model array variables) may yield to undecidable extensions of widely used theories like $\mathcal{LIA}$ (see the discussion in section 2.2.2). Given this negative result, it is mandatory to identify fragments of the quantified theories of arrays

which are on one side still decidable and on the other side sufficiently expressive.

The quantified fragment of the theory of arrays we investigate in this chapter is a sub-fragment of (2.2). Its investigation is suggested by the kind of proof obligations arising when applying the techniques presented in chapters 3, 4 and 5. In particular, the preprocessing "flattening" the formulæ we manipulate is a key step in our frameworks. As discussed in chapter 4, for example, to generate quantified invariants we enhance the interpolation-based abstraction/refinements loop underlying our framework with a special heuristic, called *term abstraction*, as discussed in section 4.1.1, aimed at searching 'good' interpolants. For the heuristic to be productive, formulæ must be subjected to 'flatness' limitations on dereferencing: only positions named by variables are allowed in dereferencing[1]. This gives the possibility to abstract out the undesired term t while simultaneously synthesizing a genuinely quantified assertion. Thus, flatness limitations constitute a key entry for some heuristics to work.

In this chapter, we leverage flatness conditions to identify a new decidable sub-fragment of (2.2). Notably, as we introduced in section 2.2.1, it is trivially true that every formula can be flattened via logical equivalences introducing extra quantifiers, i.e., by exploiting the rewriting rules

$$\phi(a(t),...) \rightsquigarrow \exists x(x = t \wedge \phi(a(x),...)) \quad \text{or} \quad \phi(a(t),...) \rightsquigarrow \forall x(x = t \rightarrow \phi(a(x),...))$$

On the other side, these rewriting rules may alter the quantifiers prefix of a formula, and it is also well-known that decidability results are sensible to the shape of quantifiers prefixes in prenex normal forms.

Flatness limitations combined with prefix limitations may introduce meaningful restrictions: this chapter shows that such restrictions can play a positive role for getting decidability. Another feature that can lead to decidability is the limitation to a single universally quantified variable in certain contexts [Habermehl et al., 2008b] and in fact we show that this kind of limitation can be usefully combined with flatness restrictions too.

The contribution of this chapter is the definition of new decidable sub-fragments of (2.2), that we called *Flat Array Properties*. We will examine Flat Array Properties of both $\mathtt{ARR}^1(\mathcal{T})$ and $\mathtt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$. Our decidability results are fully declarative and parametric in the theories $\mathcal{T}, \mathcal{T}_I, \mathcal{T}_E$. For both settings, we provide sufficient conditions on $\mathcal{T}$ and $\mathcal{T}_I, \mathcal{T}_E$ for achieving the decidability of Flat Array Properties. Such hypotheses are widely met by theories of interest in practice, like $\mathcal{LIA}$. Our decision procedures reduce the decidability of Flat

[1] Let $\phi(a[t], \dots)$ be a formula involving the term $a[t]$. The "flat" version of $\phi(a[t], \dots)$ is the equivalent formula $\exists x\ (x = t \wedge \phi(a[x], \dots))$

Array Properties to the decidability of $\mathcal{T}$-formulæ in one case and $\mathcal{T}_I$- and $\mathcal{T}_E$-formulæ in the other case.

6.1 Background notation

In this chapter we assume the notions introduced in chapter 2. Notationally, we recall that we use $\mathbf{a}$ for a tuple $\mathbf{a} = a_1, \ldots, a_{|\mathbf{a}|}$ of distinct 'array constants' (i.e., free function symbols); if $\mathbf{t} = t_1, \ldots, t_{|\mathbf{t}|}$ is a tuple of terms, the notation $\mathbf{a}(\mathbf{t})$ represents the tuple (of length $|\mathbf{a}| \cdot |\mathbf{t}|$) of terms $a_1(t_1)$, ..., $a_1(t_{|\mathbf{t}|})$, ..., $a_{|\mathbf{a}|}(t_1)$, ..., $a_{|\mathbf{a}|}(t_{|\mathbf{t}|})$.

In this chapter we will identify quantified fragments of theories as follows. Let $\mathcal{T} = (\Sigma, \mathcal{C})$ be a theory and let R be a regular expression over the alphabet $\{\exists, \forall\}$. The *R-class* of $\mathcal{T}$ is the class of Σ-formulæ comprising all and only those prenex Σ-formulæ whose prefix generates a string $Q_1 \cdots Q_n$ matched by R.

6.2 The mono-sorted case

In this section we consider the Flat Array Properties over the mono-sorted theory of arrays.

Let $\mathcal{T} = (\Sigma, \mathcal{C})$ be a theory, and $\mathtt{ARR}^1(\mathcal{T})$ be the theory obtained from $\mathcal{T}$ by adding to it infinitely many (fresh) free unary function symbols. Recall that $\mathtt{ARR}^1(\mathcal{T})$ can be undecidable even if $\mathcal{T}$ is fully decidable. We start stating the following theorem.

Theorem 6.2.1. *If the $\mathcal{T}$-satisfiability of $\exists^*\forall\exists^*$ sentences is decidable, then the $\mathtt{ARR}^1(\mathcal{T})$-satisfiability of $\exists^*\forall$-flat sentences is decidable.*

Proof. We present an algorithm, $\mathsf{SAT}_{\mathsf{MONO}}$, for deciding the satisfiability of the $\exists^*\forall$-flat fragment of $\mathtt{ARR}^1(\mathcal{T})$ (we let $\mathcal{T}$ be $(\Sigma, \mathcal{C})$). Subsequently, we show that $\mathsf{SAT}_{\mathsf{MONO}}$ is sound and complete. From the complexity viewpoint, notice that $\mathsf{SAT}_{\mathsf{MONO}}$ produces a quadratic instance of a $\exists^*\forall\exists^*$-satisfiability problem. □

6.2.1 The decision procedure $\mathsf{SAT}_{\mathsf{MONO}}$

Step I. Let

$$F := \exists \mathbf{c}\, \forall i. \psi(i, \mathbf{a}(i), \mathbf{c}, \mathbf{a}(\mathbf{c}))$$

be a $\exists^*\forall$-flat $\mathtt{ARR}^1(T)$-sentence, where ψ is a quantifier-free Σ-formula. Suppose that s is the length of $\mathbf{a}$ and t is the length of $\mathbf{c}$ (that is, $\mathbf{a} =$

$a_1, \ldots, a_s$ and $\mathbf{c} = c_1, \ldots, c_t$). Let $\mathbf{e} = \langle e_{l,m} \rangle$ $(1 \leq l \leq s,\ 1 \leq m \leq t)$ be a tuple of length $s \cdot t$ of fresh variables and consider the $\mathtt{ARR}^1(T)$-formula:

$$F_1 := \exists \mathbf{c}\, \exists \mathbf{e}\, \forall i. \psi(i, \mathbf{a}(i), \mathbf{c}, \mathbf{e}) \wedge \bigwedge_{1 \leq l \leq t} \bigwedge_{1 \leq m \leq s} a_m(c_l) = e_{l,m}$$

Step II. Build the formula (logically equivalent to F_1)

$$F_2 := \exists \mathbf{c}\, \exists \mathbf{e}\, \forall i. \left[\psi(i, \mathbf{a}(i), \mathbf{c}, \mathbf{e}) \wedge \bigwedge_{1 \leq l \leq t} (i = c_l \rightarrow \bigwedge_{1 \leq m \leq s} a_m(i) = e_{l,m}) \right]$$

Step III. Let $\mathbf{d}$ be a fresh tuple of variables of length s; check the T-satisfiability of

$$F_3 := \exists \mathbf{c}\, \exists \mathbf{e}\, \forall i\, \exists \mathbf{d}. \left[\psi(i, \mathbf{d}, \mathbf{c}, \mathbf{e}) \wedge \bigwedge_{1 \leq l \leq t} (i = c_l \rightarrow \bigwedge_{1 \leq m \leq s} d_m = e_{l,m}) \right]$$

6.2.2 Correctness and completeness

$\mathsf{SAT}_{\mathsf{MONO}}$ transforms an $\mathtt{ARR}^1(\mathcal{T})$-formula F into an equisatisfiable $\mathcal{T}$-formula F_3 belonging to the $\exists^*\forall\exists^*$ fragment. More precisely, it holds that F, F_1 and F_2 are equivalent formulæ, because

$$\bigwedge_{1 \leq l \leq t} \forall i. (i = c_l \rightarrow \bigwedge_{1 \leq m \leq s} a_m(i) = e_{l,m}) \equiv \bigwedge_{1 \leq l \leq t} \bigwedge_{1 \leq m \leq s} a_m(c_l) = e_{l,m}$$

From F_2 to F_3 and back, satisfiability is preserved because F_2 is the Skolemization of F_3, where the existentially quantified variables $\mathbf{d} = d_1, \ldots, d_s$ are substituted with the free unary function symbols $\mathbf{a} = a_1, \ldots a_s$. In the above proof, it is essential that F is flat and that only one universally quantified variable occurs in it: these features are precisely the features needed for the formula F_2 to come from the Skolemization of F_3.

Since $\mathcal{LIA}$ is decidable (via quantifier elimination), we get in particular that

Corollary 6.2.1. *The $\mathtt{ARR}^1(\mathcal{LIA})$-satisfiability of $\exists^*\forall$-flat sentences is decidable.*

6.3 The multi-sorted case

We are now considering a multi-sorted theory of arrays parametric in the theories specifying constraints over indexes and elements of the arrays, i.e., a theory of the kind $\texttt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$. Recall from section 2.2 that $\mathcal{T}_I = (\Sigma_I, \mathcal{C}_I)$ and $\mathcal{T}_E = (\Sigma_E, \mathcal{C}_E)$; we assume that Σ_I and Σ_E are disjoint and for simplicity, we let both signatures be mono-sorted (but extending our results to many-sorted $\mathcal{T}_E$ is quite straightforward): `INDEX` is the unique sort of $\mathcal{T}_I$ and `ELEM` the unique sort of $\mathcal{T}_E$. Now the theory $\texttt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$ *of arrays over* $\mathcal{T}_I$ *and* $\mathcal{T}_E$ is obtained from the union of $\Sigma_I \cup \Sigma_E$ by adding to it infinitely many (fresh) free unary function symbols (these new function symbols will have domain sort `INDEX` and codomain sort `ELEM`). The models of $\texttt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$ are the structures whose reducts to the symbols of sorts `INDEX` and `ELEM` are models of $\mathcal{T}_I$ and $\mathcal{T}_E$, respectively.

Consider now an atomic formula $P(t_1, \ldots, t_n)$ in the language of $\texttt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$. Since the predicate symbols of $\texttt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$ are from $\Sigma_I \cup \Sigma_E$ and $\Sigma_I \cap \Sigma_E = \emptyset$, P belongs either to Σ_I or to Σ_E; in the former case all terms t_i are Σ_I-terms (notice in fact that to produce a term of sort `INDEX` one must use only Σ_I-symbols) and in the latter case, all terms t_i have sort `ELEM`. We say that $P(t_1, \ldots, t_n)$ is an `INDEX`-*atom* in the former case and that it is an `ELEM`-*atom* in the latter case.

When dealing with $\texttt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$, we shall limit ourselves to quantified variables of sort `INDEX`: this limitation is justified by the application we target in this thesis, i.e., verification of programs with arrays[2].

Definition 6.3.1 (Monic sentence)**.** *A sentence in the language of* $\texttt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$ *is said to be* monic *iff it is in prenex form and every* `INDEX` *atom occurring in it contains at most one universally quantified variable.*

Example 6.3.1. Consider the following sentences:

$(I)\ \forall i.\, a(i) = i;$ $(II)\ \forall i_1 \forall i_2.\, (i_1 \leq i_2 \rightarrow a(i_1) \leq a(i_2));$
$(III)\ \exists i_1 \exists i_2.\, (i_1 \leq i_2 \wedge a(i_1) \not\leq a(i_2));$ $(IV)\ \forall i_1 \forall i_2.\, a(i_1) = a(i_2);$
$(V)\ \forall i.\, (D_2(i) \rightarrow a(i) = 0);$ $(VI)\ \exists i\ \forall j.\, (a_1(j) < a_2(3i)).$

The flat formula (I) is not well-typed, hence it is not allowed in $\texttt{ARR}^2(\mathcal{LIA}, \mathcal{LIA})$; however, it is allowed in $\texttt{ARR}^1(\mathcal{LIA})$. Formula (II) expresses the fact that the array a is sorted: it is flat but not monic (because of the atom $i_1 \leq i_2$). On

[2]Topmost existentially quantified variables of sort `ELEM` can be modeled by enriching $\mathcal{T}_E$ with free constants.

the contrary, its negation (III) is flat and monic (because i_1, i_2 are now existentially quantified). Formula (IV) expresses that the array a is constant; it is flat and monic (notice that the universally quantified variables i_1, i_2 both occur in $a(i_1) = a(i_2)$ but the latter is an ELEM atom). Formula (V) expresses that a is initialized so to have all even positions equal to 0: it is monic and flat. Formula (VI) is monic but not flat because of the term $a_2(3i)$ occurring in it; however, in $3i$ no universally quantified variable occurs, so it is possible to produce by flattening the following sentence

$$\exists i\, \exists i'\, \forall j\ (i' = 3i \wedge a_1(j) < a_2(i'))$$

which is logically equivalent to (VI), it is flat and still lies in the $\exists^*\forall$-class. Finally, as a more complicated example, notice that the following sentence

$$\exists k\, \forall i.\ \begin{pmatrix} D_2(k) \wedge a(k) = \text{`}\backslash\mathtt{0}\text{`} \wedge \\ (D_2(i) \wedge i < k \rightarrow a(i) = \text{`}\mathtt{b}\text{`}) \wedge \\ (\neg D_2(i) \wedge i < k \rightarrow a(i) = \text{`}\mathtt{c}\text{`}) \end{pmatrix}$$

is monic and flat: it says that a represents a string of the kind $(\mathtt{bc})^*$.

Theorem 6.3.1. *If $\mathcal{T}_I$-satisfiability of $\exists^*\forall$-sentences is decidable, then $\mathtt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$-satisfiability of $\exists^*\forall^*$-monic-flat sentences is decidable.*

Proof. As we did for $\mathsf{SAT}_{\mathsf{MONO}}$, we give a decision procedure, $\mathsf{SAT}_{\mathsf{MULTI}}$, for the $\exists^*\forall^*$-monic-flat fragment of $\mathtt{ARR}^2(T_I, T_E)$. Since the procedure is complex, we divide our exposition in different phases. We summarize again here some high level information, then we formally introduce the procedure in section 6.3.1. Correctness and completeness of $\mathsf{SAT}_{\mathsf{MULTI}}$ are split into two lemmas (Lemmas 6.3.2 and 6.3.1) to be proved in section 6.3.2 below.

First (Step I), the procedure *guesses* the sets (called 'types') of relevant INDEX atoms satisfied in a model to be built. Subsequently (Step II) it introduces a witness existential variable for each type together with the constraint that guessed types are exhaustive. Finally (Step III, IV and V) the procedure applies combination techniques for purification. □

Theorem 6.3.1 reduces the $\mathtt{ARR}^2(T_I, T_E)$-satisfiability of $\exists^*\forall^*$-monic-flat sentences to the T_I-satisfiability of $\exists^*\forall$-sentences. We give here an informal account of the main argument we use in the proof. The fact that the formulæ to be tested for satisfiability are monic is essential[3] and we make use of this hy-

[3]Undecidability arises otherwise, see the discussion of section 2.2.2 for a reduction to reachability problems of Minsky machines.

pothesis by introducing witnesses for the realized *unary* types. The notion of a type is commonly used in model theory; we adapt it to our context by defining a type to be a *maximal consistent set* of `INDEX` literals occurring in the formula to be tested for satisfiability. In other words: in every model, every element from the support of the interpretation of the `INDEX` sort satisfies a maximal consistent set of such `INDEX` literals; the latter, modulo renaming, are of the kind $L(i, \mathbf{c})$ (only *one* free variable occurs here by the monicity hypothesis, the $\mathbf{c}$ are free constants coming from the Skolemization of the outermost existential quantifiers). The satisfiability algorithm guesses in advanced which types M are realized (i.e. satisfied), it introduces for each of them a witness constant b_M, it takes the conjunction of the original formula with the literals $L(b_M, \mathbf{c})$ for $L \in M$ (varying M) and with a universal `INDEX` formula saying that only the guessed types are realized. Then, the single universal quantifier of the original formula is instantiated over all constants. The final part of the algorithm follows some Nelson-Oppen like combination schema in order to separately test the `INDEX` and the `ELEM` components for satisfiability. We point out, once again, that the above machinery works because we need to care about unary types only; if we had to deal with non-monic formulæ, we were in trouble: guessing binary types (i.e. maximal consistent sets of two-variables literals) would not be sufficient, as one should also guess ternary types to match, e.g., the second components and the first components of binary types, etc., making the combinatorics out of control.

Notably, by considering the special case of formulæ in which `ELEM` atoms do not occur, Theorem 6.3.1 has the following corollary concerning only $\mathcal{T}_I$.

Corollary 6.3.1. *If $\mathcal{T}_I$-satisfiability of the $\exists^*\forall$-sentences is decidable, then $\mathcal{T}_I$-satisfiability of $\exists^*\forall^*$-monic-flat sentences is decidable.*

This is because by help of (rather expensive) Boolean manipulations one can check directly that $\exists^*\forall^*$-monic-flat $\mathcal{T}_I$-sentences are equivalent to disjunctions of $\exists^*\forall$ $\mathcal{T}_I$-sentences. In other words, the notion of being monic becomes interesting only in presence of `ELEM` atoms.

6.3.1 The decision procedure $\mathsf{SAT}_{\mathsf{MULTI}}$

The algorithm is non-deterministic: the input formula is satisfiable iff we can guess suitable data $\mathfrak{T}, \mathfrak{B}$ so that the formulæ F_I, F_E below are satisfiable.

Step I. Let F be a $\exists^*\forall^*$-monic-flat formula; let it be

$$F := \exists \mathbf{c}\, \forall \mathbf{i}. \psi(\mathbf{i}, \mathbf{a}(\mathbf{i}), \mathbf{c}, \mathbf{a}(\mathbf{c})),$$

(where as usual ψ is a $\mathcal{T}_I \cup \mathcal{T}_E$-quantifier-free formula). Suppose $\mathbf{a} = a_1, \ldots, a_s$, $\mathbf{i} = i_1, \ldots, i_n$ and $\mathbf{c} = c_1, \ldots, c_t$. Consider the set (notice that all atoms in K are Σ_I-atoms and have just one free variable because F is monic)

$$K = \{A(x, \mathbf{c}) \mid A(i_k, \mathbf{c}) \text{ is an } \texttt{INDEX} \text{ atom of } F\}_{1 \leq k \leq n} \cup \{x = c_l\}_{1 \leq l \leq t}$$

Let us call *type* a set of literals M such that: (i) each literal of M is an atom in K or its negation; (ii) for all $A(x, \mathbf{c}) \in K$, either $A(x, \mathbf{c}) \in M$ or $\neg A(x, \mathbf{c}) \in M$. Guess a set $\mathfrak{T} = \{M_1, \ldots, M_q\}$ of types.

Step II. Let $\mathbf{b} = b_1, \ldots, b_q$ be a tuple of new variables of sort `INDEX` and let

$$F_1 := \exists \mathbf{b}\, \exists \mathbf{c} \left[\begin{array}{l} \forall x. \left(\bigvee_{j=1}^{q} \bigwedge_{L \in M_j} L(x, \mathbf{c}) \right) \wedge \\ \bigwedge_{j=1}^{q} \bigwedge_{L \in M_j} L(b_j, \mathbf{c}) \wedge \\ \bigwedge_{\sigma : \mathbf{i} \to \mathbf{b}} \psi(\mathbf{i}\sigma, \mathbf{a}(\mathbf{i}\sigma), \mathbf{c}, \mathbf{a}(\mathbf{c})) \end{array} \right]$$

where $\mathbf{i}\sigma$ is the tuple of terms $\sigma(i_1), \ldots, \sigma(i_n)$.

Step III. Let $\mathbf{e} = \langle e_{l,m} \rangle$ $(1 \leq l \leq s,\ 1 \leq m \leq t + q)$ be a tuple of length $s \cdot (t + q)$ of free constants of sort `ELEM`. Consider the formula

$$F_2 := \exists \mathbf{b}\, \exists \mathbf{c} \left[\begin{array}{l} \forall x. \left(\bigvee_{j=1}^{q} \bigwedge_{L \in M_j} L(x, \mathbf{c}) \right) \wedge \\ \bigwedge_{j=1}^{q} \bigwedge_{L \in M_j} L(b_j, \mathbf{c}) \wedge \\ \bar{\psi}(\mathbf{b}, \mathbf{c}, \mathbf{e}) \wedge \\ \bigwedge_{d_m, d_n \in \mathbf{b} * \mathbf{c}} \bigwedge_{l=1}^{s} (d_m = d_n \to e_{l,m} = e_{l,n}) \end{array} \right]$$

where $\mathbf{b} * \mathbf{c} := d_1, \ldots, d_{q+t}$ is the concatenation of the tuples $\mathbf{b}$ and $\mathbf{c}$ and

$\bar{\psi}(\mathbf{b}, \mathbf{c}, \mathbf{e})$ is obtained from

$$\bigwedge_{\sigma:\mathbf{i}\to\mathbf{b}} \psi(\mathbf{i}\sigma, \mathbf{a}(\mathbf{i}\sigma), \mathbf{c}, \mathbf{a}(\mathbf{c}))$$

by substituting each term in the tuple $\mathbf{a}(\mathbf{b}) * \mathbf{a}(\mathbf{c})$ with the constant occupying the corresponding position in the tuple $\mathbf{e}$.

Step IV. Let $\mathfrak{B}$ a full Boolean satisfying assignment for the atoms of the formula

$$F_3 := \bar{\psi}(\mathbf{b}, \mathbf{c}, \mathbf{e}) \ \wedge \bigwedge_{d_m, d_n \in \mathbf{b}*\mathbf{c}} \bigwedge_{l=1}^{s} (d_m = d_n \to e_{l,m} = e_{l,n})$$

and let $\bar{\psi}_I(\mathbf{b}, \mathbf{c}), \bar{\psi}_E(\mathbf{e})$ be the (conjunction of the) sets of literals of sort `INDEX` and `ELEM`, respectively, induced by $\mathfrak{B}$.

Step V. Check the $\mathcal{T}_I$-satisfiability of

$$F_I := \exists \mathbf{b}\, \exists \mathbf{c}. \left[\forall x. \left(\bigvee_{j=1}^{q} \bigwedge_{L \in M_j} L(x, \mathbf{c}) \right) \wedge \bigwedge_{j=1}^{q} \bigwedge_{L \in M_j} L(b_j, \mathbf{c}) \wedge \bar{\psi}_I(\mathbf{b}, \mathbf{c}) \right]$$

and the $\mathcal{T}_E$-satisfiability of

$$F_E := \bar{\psi}_E(\mathbf{e})$$

Notice that F_I is an $\exists^*\forall$-sentence; F_E is ground and the $\mathcal{T}_E$-satisfiability of F_E (considering the $\mathbf{e}$ as variables instead of as free constants) is decidable because we assumed that all the theories we consider in this thesis have quantifier-free fragments decidable for satisfiability. The procedure $\mathsf{SAT}_{\mathsf{MULTI}}$ returns SAT if both satisfiability tests are successful.

6.3.2 Correctness and Completeness

Before proving correctness and completeness, we introduce useful notation. We use letters $\tilde{b}, \tilde{c}, \ldots$ for elements from the support of a model; notation $\tilde{\mathbf{b}}, \tilde{\mathbf{c}}, \ldots$ is used for tuples (possibly with repetitions) of such elements. For a formula $\varphi(\mathbf{c})$ containing the free variables $\mathbf{c} := c_1, \ldots, c_n$ and for a tuple of elements $\tilde{\mathbf{c}} := \tilde{c}_1, \ldots, \tilde{c}_n$ from the support of a model $\mathcal{M}$, $\mathcal{M} \models \varphi(\tilde{\mathbf{c}})$ means that $\varphi(\mathbf{c})$ is true in $\mathcal{M}$ under the assignment mapping the $\mathbf{c}$ to the $\tilde{\mathbf{c}}$.

Below, we assume that F is the $\exists^*\forall^*$-monic-flat formula

$$F := \exists \mathbf{c}\, \forall \mathbf{i}. \psi(\mathbf{i}, \mathbf{a}(\mathbf{i}), \mathbf{c}, \mathbf{a}(\mathbf{c}));$$

the formulæ F_1, F_2, F_3, F_I, F_E are as described in the decision procedure $\mathsf{SAT}_{\mathsf{MULTI}}$ of section 6.3.1.

Lemma 6.3.1 (Completeness of $\mathsf{SAT}_{\mathsf{MULTI}}$)**.** *If F is $\mathtt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$-satisfiable, then it is possible to choose the set $\mathfrak{T}$ and the Boolean assignment $\mathfrak{B}$ so that F_I is $\mathcal{T}_I$-satisfiable and F_E is $\mathcal{T}_E$-satisfiable.*

Proof. Let $\mathcal{M}$ be a model of F. We have $\mathcal{M} \models \forall \mathbf{i}. \psi(\mathbf{i}, \mathbf{a}(\mathbf{i}), \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}}))$ for suitable $\tilde{\mathbf{c}}$ from $\mathtt{INDEX}^{\mathcal{M}}$.

A type M is *realized* in $\mathcal{M}$ iff there is some $\tilde{b} \in \mathtt{INDEX}^{\mathcal{M}}$ such that $\mathcal{M} \models \bigwedge_{L \in M} L(\tilde{b}, \tilde{\mathbf{c}})$ (we say in this case that $\tilde{b}$ realizes M).[4] We take $\mathfrak{T}$ to be the set of types realized in $\mathcal{M}$; if $\mathfrak{T} = \{M_1, \ldots, M_q\}$, we pick a tuple $\tilde{\mathbf{b}} = \tilde{b}_1, \ldots, \tilde{b}_q$ from $\mathtt{INDEX}^{\mathcal{M}}$ realizing them. By assigning precisely this tuple to the variables $\mathbf{b}$ of F_1, we get

$$\begin{aligned} \mathcal{M} \models\ & \forall x. \left(\bigvee_{j=1}^{q} \bigwedge_{L \in M_j} L(x, \tilde{\mathbf{c}}) \right) \wedge \\ & \bigwedge_{j=1}^{q} \bigwedge_{L \in M_j} L(\tilde{b}_j, \tilde{\mathbf{c}}) \wedge \\ & \bigwedge_{\sigma : \mathbf{i} \to \tilde{\mathbf{b}}} \psi(\mathbf{i}\sigma, \mathbf{a}(\mathbf{i}\sigma), \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}})) \end{aligned}$$

(this formula is F_1 without the outermost existential quantifiers and with $\mathbf{c}, \mathbf{b}$ replaced by - the names of - $\tilde{\mathbf{c}}, \tilde{\mathbf{b}}$). If we furthermore let the tuple $\tilde{\mathbf{e}}$ be the

[4] Notice that this type realization notion is relative to the choice of the elements $\tilde{\mathbf{c}}$ assigned to the $\mathbf{c}$.

tuple of the elements denoted by the terms $\mathbf{a}[\tilde{\mathbf{c}}] * \mathbf{a}[\tilde{\mathbf{b}}]$, we get[5]

$$\begin{aligned}\mathcal{M} \models\ \forall x.\left(\bigvee_{j=1}^{q}\bigwedge_{L\in M_j} L(x,\tilde{\mathbf{c}})\right)\ \wedge\\ \bigwedge_{j=1}^{q}\bigwedge_{L\in M_j} L(\tilde{b}_j,\tilde{\mathbf{c}})\ \wedge\\ \bar{\psi}(\tilde{\mathbf{b}},\tilde{\mathbf{c}},\tilde{\mathbf{e}})\ \wedge\\ \bigwedge_{\tilde{d}_m,\tilde{d}_n\in\tilde{\mathbf{b}}*\tilde{\mathbf{c}}}\ \bigwedge_{l=1}^{s}(\tilde{d}_m=\tilde{d}_n\rightarrow\tilde{e}_{l,m}=\tilde{e}_{l,n})\end{aligned}$$

as well. Now we can get our $\mathfrak{B}$ just by collecting the truth-values of the relevant `INDEX` and `ELEM` atoms involved in the above formula; by construction, it is clear that F_I and F_E become both true. □

Lemma 6.3.2 (Soundness of $\mathsf{SAT}_{\mathsf{MULTI}}$)**.** *If there exist* $\mathfrak{T} := \{M_1,\ldots,M_q\}$ *and* $\mathfrak{B}$ *such that* F_I *is* $\mathcal{T}_I$*-satisfiable and* F_E *is* $\mathcal{T}_E$*-satisfiable, then* F *is* $\mathtt{ARR}^2(\mathcal{T}_I,\mathcal{T}_E)$*-satisfiable.*

Proof. Suppose we are given a set of types $\mathfrak{T} = \{M_1,\ldots,M_q\}$ and a Boolean assignment $\mathfrak{B}$ such that there exists two models $\mathcal{M}_I, \mathcal{M}_E$ of $\mathcal{T}_I, \mathcal{T}_E$, respectively, such that $\mathcal{M}_I \models F_I$ and $\mathcal{M}_E \models F_E$. From the fact that F_I is true in $\mathcal{M}_I$, it follows that there are elements $\tilde{\mathbf{c}}, \tilde{\mathbf{b}}$ from $\mathtt{INDEX}^{\mathcal{M}_I}$ such that

$$\mathcal{M}_I \models \forall x.\left(\bigvee_{j=1}^{q}\bigwedge_{L\in M_j} L(x,\tilde{\mathbf{c}})\right) \wedge \bigwedge_{j=1}^{q}\bigwedge_{L\in M_j} L(\tilde{b}_j,\tilde{\mathbf{c}}) \wedge \tilde{\psi}_I(\tilde{\mathbf{b}},\tilde{\mathbf{c}})\ . \tag{6.1}$$

In particular,

$$\mathcal{M}_I \models \bigwedge_{L\in M_j} L(\tilde{b}_j,\tilde{\mathbf{c}})$$

holds for every $M_j \in \mathfrak{T}$. Thus, each $M_j \in \mathfrak{T}$ is associated with an element $\tilde{b}_j \in \mathtt{INDEX}^{\mathcal{M}_I}$ that realizes it, while

$$\mathcal{M}_I \models \forall x.\left(\bigvee_{j=1}^{q}\bigwedge_{L\in M_j} L(x,\tilde{\mathbf{c}})\right) \tag{6.2}$$

[5] In particular, $\mathcal{M} \models \forall x.(\bigvee_{j=1}^{q}\bigwedge_{L\in M_j} L(x,\tilde{\mathbf{c}}))$ says that at most $M_1,\ldots,M_q$ are realized and the second conjunct says that in fact $M_1,\ldots,M_q$ are realized (by $\tilde{b}_1,\ldots,\tilde{b}_q$, respectively).

implies that every $\tilde{z} \in \mathtt{INDEX}^{\mathcal{M}_I}$ realizes some $M_j \in \mathfrak{T}$ (see the proof of the previous Lemma for the definition of type realization). We introduce the following notation: given two elements $\tilde{z}_1, \tilde{z}_2 \in \mathtt{INDEX}^{\mathcal{M}_I}$, $\tilde{z}_1 \approx \tilde{z}_2$ holds iff $\tilde{z}_1$ and $\tilde{z}_2$ realize the same type. Thus, for every $\tilde{z} \in \mathtt{INDEX}^{\mathcal{M}_I}$ there is a (unique because types are mutually inconsistent) $\tilde{b}_j \in \tilde{\mathbf{b}}$ such that $\tilde{z} \approx \tilde{b}_j$. We call this b_j the *representative* of $\tilde{z}$.

Now, since $\mathcal{M}_E \models F_E$, there are elements $\tilde{\mathbf{e}} \in \mathtt{ELEM}^{\mathcal{M}_E}$ such that (once they are used to interpret the constants $\mathbf{e}$) we have

$$\mathcal{M}_E \models \bar{\psi}_E(\tilde{\mathbf{e}}) \ . \tag{6.3}$$

To get a model $\mathcal{M}$ for $\mathtt{ARR}^2(\mathcal{T}_I, \mathcal{T}_E)$ we need only to interpret the function symbols $\mathbf{a} = a_1, \ldots, a_s$ as functions from $\mathtt{INDEX}^{\mathcal{M}_I}$ into $\mathtt{ELEM}^{\mathcal{M}_E}$. Before doing that, let us observe that, because of our choice of $\mathfrak{B}$, we have that $\bar{\psi}_I(\mathbf{b}, \mathbf{c}) \wedge \bar{\psi}_E(\mathbf{e}) \rightarrow F_3$ is a tautology. Recalling the definition of F_3 from Step IV of the procedure $\mathsf{SAT}_{\mathsf{MULTI}}$, this means that (independently on how we define the interpretation of the symbols $\mathbf{a}$ not occurring in F_3) we shall have

$$\mathcal{M} \models \bar{\psi}(\tilde{\mathbf{b}}, \tilde{\mathbf{c}}, \tilde{\mathbf{e}}) \ \wedge \bigwedge_{\tilde{d}_m, \tilde{d}_n \in \tilde{\mathbf{b}} * \tilde{\mathbf{c}}} \bigwedge_{l=1}^{s} (\tilde{d}_m = \tilde{d}_n \rightarrow \tilde{e}_{l,m} = \tilde{e}_{l,n}) \ . \tag{6.4}$$

For every $l = 1, \ldots, s$ and for every $\tilde{d}_m \in \tilde{\mathbf{b}} * \tilde{\mathbf{c}}$ we put

$$a_l^{\mathcal{M}}(\tilde{d}_m) := \tilde{e}_{l,m} \ . \tag{6.5}$$

By (6.4), this definition gives a partial function. To make it total, for any other $\tilde{z}$ (i.e. $\tilde{z} \notin \tilde{\mathbf{b}} * \tilde{\mathbf{c}}$) pick the representative $\tilde{b}_j$ of $\tilde{z}$, and define

$$a_l^{\mathcal{M}}(\tilde{z}) := a_l^{\mathcal{M}}(\tilde{b}_j) \ . \tag{6.6}$$

We *claim* that we have, for every $\tilde{z}_1, \tilde{z}_2 \in \mathtt{INDEX}^{\mathcal{M}_I}$

$$\tilde{z}_1 \approx \tilde{z}_2 \quad \Rightarrow \quad a_l^{\mathcal{M}}(\tilde{z}_1) = a_l^{\mathcal{M}}(\tilde{z}_2) \ . \tag{6.7}$$

To prove the claim, it is sufficient to show that, if $\tilde{b}_j$ is the representative of $\tilde{z}$, then $a_l^{\mathcal{M}}(\tilde{z}) = a_l^{\mathcal{M}}(\tilde{b}_j)$. This is obvious if $\tilde{z} \notin \tilde{\mathbf{b}} * \tilde{\mathbf{c}}$ and if $\tilde{z} \in \tilde{\mathbf{b}} * \tilde{\mathbf{c}}$, we only have to check the case in which $\tilde{z}$ is some $\tilde{c}_l \in \tilde{\mathbf{c}}$. However, since $x = c_l$ is among the atoms contributing to the definition of a type (see Step I of the procedure $\mathsf{SAT}_{\mathsf{MULTI}}$), it follows that the representative $\tilde{b}_j$ of $\tilde{c}_l$ satisfies the formula $x = \tilde{c}_l$ (because the latter is trivially satisfied by $\tilde{c}_l$) and hence we have that $\tilde{b}_j = \tilde{c}_l$.

By (6.4) and (6.5), it follows that $a_l^{\mathcal{M}}(\tilde{c}_j) = a_l^{\mathcal{M}}(\tilde{b}_j)$. This ends the proof of the claim.

It remains to prove that $\mathcal{M}$ is a model of F, i.e. that we have

$$\mathcal{M} \models \forall \mathbf{i}.\psi(\mathbf{i}, \mathbf{a}(\mathbf{i}), \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}})) \ . \tag{6.8}$$

First notice that, by (6.5),(6.4) and by the definition of $\bar{\psi}(\mathbf{b}, \mathbf{c}, \mathbf{e})$ (see STEP III of the procedure $\mathsf{SAT}_{\mathsf{MULTI}}$), we have[6]

$$\mathcal{M} \models \bigwedge_{\sigma:\mathbf{i}\to\tilde{\mathbf{b}}} \psi(\mathbf{i}\sigma, \mathbf{a}(\mathbf{i}\sigma), \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}})) \ . \tag{6.9}$$

Let τ be the map that associates with every $\tilde{z}$ its representative $\tilde{b}_j \in \tilde{\mathbf{b}}$; it is sufficient to show that for every $\tilde{\mathbf{z}} = \tilde{z}_1, \ldots, \tilde{z}_n$ from $\mathtt{INDEX}^{\mathcal{M}}$,[7] we have, for every atom $A(\mathbf{i}, \mathbf{a}(\mathbf{i}), \mathbf{c}, \mathbf{a}(\mathbf{c}))$ occurring in $\psi(\mathbf{i}, \mathbf{a}(\mathbf{i}), \mathbf{c}, \mathbf{a}(\mathbf{c}))$

$$\mathcal{M} \models A(\tilde{\mathbf{z}}, \mathbf{a}(\tilde{\mathbf{z}}), \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}})) \leftrightarrow A(\tilde{\mathbf{z}}\tau, \mathbf{a}(\tilde{\mathbf{z}}\tau), \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}})) \tag{6.10}$$

(then (6.8) follows from (6.9) and (6.10) by induction on the number of Boolean connectives in ψ, taking for every assignment $\mathbf{i} \mapsto \tilde{\mathbf{z}}$ the conjunct σ corresponding to $\mathbf{i} \mapsto \tilde{\mathbf{z}} \mapsto \tilde{\mathbf{z}}\tau$). In turn, (6.10) is a special case of the following more general fact: if $\tilde{\mathbf{z}}$ and $\tilde{\mathbf{z}}'$ have length n and we have $\tilde{z}_i \approx \tilde{z}'_i$ (for every $i = 1, \ldots, n$), then

$$\mathcal{M} \models A(\tilde{\mathbf{z}}, \mathbf{a}(\tilde{\mathbf{z}}), \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}})) \leftrightarrow A(\tilde{\mathbf{z}}', \mathbf{a}(\tilde{\mathbf{z}}'), \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}})) \tag{6.11}$$

for every atom A occurring in ψ. However, (6.11) holds for `ELEM` atoms thanks to (6.7) and for `INDEX` atoms due to the fact that $\tilde{z}_i, \tilde{z}'_i$ realize the same type and the input formula $F := \exists \mathbf{c}\, \forall \mathbf{i}.\psi(\mathbf{i}, \mathbf{a}(\mathbf{i}), \mathbf{c}, \mathbf{a}(\mathbf{c}))$ is monic. □

6.3.3 Complexity Analysis.

Theorem 6.3.1 applies to $\mathtt{ARR}^2(\mathcal{LIA}, \mathcal{LIA})$ because $\mathcal{LIA}$ admits quantifier elimination. For this theory, we can determine complexity upper and lower bounds:

Theorem 6.3.2. $\mathtt{ARR}^2(\mathcal{LIA}, \mathcal{LIA})$*-satisfiability of* $\exists^*\forall^*$*-monic-flat sentences is* NEXPTIME*-complete.*

[6] Since types are pairwise inconsistent, the elements $\tilde{\mathbf{b}}$ are in bijective correspondence to the variables $\mathbf{b}$, hence we can freely suppose that the maps σ indexing the big conjunct of (6.9) have codomain $\tilde{\mathbf{b}}$.

[7] Recall that n is the length of the tuple $\mathbf{i}$. Here $\tilde{\mathbf{z}}$ ranges over all possible tuples of elements that can be assigned to the tuple of variables $\mathbf{i}$.

The proof is split into the two lemmas below, giving the lower and upper bound required.

Lemma 6.3.3 (Lower Bound). $\mathtt{ARR}^2(\mathcal{LIA}, \mathcal{LIA})$*-satisfiability of* $\exists^*\forall^*$*-monic-flat sentences is* NExpTime*-hard.*

Proof. First, we introduce the bounded version of the domino problem used in the reduction. A *domino system* is a triple $\mathcal{D} = (D, H, V)$, where D is a finite set of *domino types* and $H, V \subseteq D \times D$ are the horizontal and vertical matching conditions. Let $\mathcal{D}$ be a domino system and $I = d_0, \ldots, d_{n-1} \in D^*$ an *initial condition*, i.e. a sequence of domino types of length $n > 0$. A mapping $\tau : \{0, \ldots, 2^{n+1} - 1\} \times \{0, \ldots, 2^{n+1} - 1\} \to D$ is a 2^{n+1}*-bounded solution of* $\mathcal{D}$ *respecting the initial condition* I iff, for all $x, y < 2^{n+1}$, the following holds:

- if $\tau(x, y) = d$ and $\tau(x \oplus_{2^{n+1}} 1, y) = d'$, then $(d, d') \in H$;
- if $\tau(x, y) = d$ and $\tau(x, y \oplus_{2^{n+1}} 1) = d'$, then $(d, d') \in V$;
- $\tau(i, 0) = d_i$ for $i < n$;

where $\oplus_{2^{n+1}}$ denotes addition modulo 2^{n+1}.

It is well-known [Börger et al., 1997, Lewis, 1978] that there is a domino system $\mathcal{D} = (D, H, V)$ such that the following problem is NExpTime-hard: given an initial condition $I = d_0, \ldots, d_{n-1} \in D^*$, does $\mathcal{D}$ have a 2^{n+1}-bounded solution respecting I or not?

We show that this problem can be reduced in polynomial time to satisfiability of $\exists^*\forall^*$-flat and simple sentences in $\mathtt{ARR}^2(\mathcal{LIA}, \mathcal{LIA})$.

Let us associate (in an injective way) with every element $d \in D$ a numeral (we call this numeral again d for simplicity[8]); we shall use just one array variable, to be called a.

Let $p_0, \ldots, p_n, q_0, \ldots, q_n$ be distinct pairwise co-prime numbers. We underline that $p_0, \ldots, p_n, q_0, \ldots, q_n$ can be computed in time polynomial in n and that polynomially many bits are needed to represent them and, as a consequence, also the divisibility predicates $D_{p_0}, \ldots, D_{p_n}, D_{q_0}, \ldots, D_{q_n}$ (to see that this is the case, one can use the well-known bound, proved by Rosser in [Rosser,

[8] A numeral is a ground term of the kind $1 + \cdots + 1$, i.e. a ground term canonically representing a number. The argument we use works also for weaker theories like $\mathtt{ARR}^2(\mathcal{LIA}, Eq)$, where Eq is the pure identity theory in a language containing infinitely many constants constrained to be distinct.

1939] - see also [Bach and Shallit, 1996], saying that the N-th prime number is less than $N \log N + 2N \log\log N$, for all $N > 3$).[9]

We say a natural number i *represents* the point of coordinates $(x, y) \in [0, 2^{n+1} - 1] \times [0, 2^{n+1} - 1]$ iff for all $k = 0, \ldots, n$, we have that

(i) $D_{p_k}(i)$ holds iff the k-th bit of the binary representation of x is 0;

(ii) $D_{q_k}(i)$ holds iff the k-th bit of the binary representation of y is 0.

Of course, the same (x, y) can be represented in many ways, but at least one representative number exists by the Chinese Reminder Theorem.

We now introduce the following abbreviations:

- $H_E(e, e')$ stands for $\bigvee_{(d,d')\in H}(e = d \wedge e' = d')$;
- $V_E(e, e')$ stands for $\bigvee_{(d,d')\in V}(e = d \wedge e' = d')$;
- $H_I(i, i')$ stands for the conjunction of $\bigwedge_{k=0}^{n}(D_{q_k}(i) \leftrightarrow D_{q_k}(i'))$ with

$$\left(\bigwedge_{k=0}^{n}(\neg D_{p_k}(i) \wedge D_{p_k}(i'))\right) \vee \bigvee_{k=0}^{n}\left(\begin{array}{l}\bigwedge_{l>k}(D_{p_l}(i) \leftrightarrow D_{p_l}(i')) \wedge \\ D_{p_k}(i) \wedge \neg D_{p_k}(i') \wedge \\ \bigwedge_{l<k}(\neg D_{p_l}(i) \wedge D_{p_l}(i'))\end{array}\right)$$

- $V_I(i, i')$ stands for the conjunction of $\bigwedge_{k=0}^{n}(D_{p_k}(i) \leftrightarrow D_{p_k}(i'))$ with

$$\left(\bigwedge_{k=0}^{n}(\neg D_{q_k}(i) \wedge D_{q_k}(i'))\right) \vee \bigvee_{k=0}^{n}\left(\begin{array}{l}\bigwedge_{l>k}(D_{q_l}(i) \leftrightarrow D_{q_l}(i')) \wedge \\ D_{q_k}(i) \wedge \neg D_{q_k}(i') \wedge \\ \bigwedge_{l<k}(\neg D_{q_l}(i) \wedge D_{q_l}(i'))\end{array}\right)$$

Thus, $H_I(i, i')$ holds iff i represents (x, y), i' represents (x', y') and we have $y = y'$ and $x' = x \oplus_{2^{n+1}} 1$. Similarly, $V_I(i, i')$ holds iff i represents (x, y), i' represents (x', y') and we have $x = x'$ and $y' = y \oplus_{2^{n+1}} 1$.

[9] For our purposes, the following elementary argument would be sufficient as well, because it gives a formula for a direct polynomial computation (under logarithmic cost criterion). Define $h(2) := 2$ and $h(n + 1) := 1 + \prod_{m<n} h(m)$; it is clear that if $k_1 < k_2$, then $h(k_1)$ and $h(k_2)$ are co-prime, because the reminder of the division of $h(k_2)$ by every factor of $h(k_1)$ is 1. Also, we easily get $h(n) \leq n!$ by induction: indeed, $h(2) \leq 2!$ and $h(n+1) \leq 1 + \prod_{m\leq n} h(m) \leq 1 + n \cdot n! \leq (n + 1)!$.

We introduce abbreviations $P_{0,0}(i), \ldots, P_{n-1,0}(i)$ to express the fact that i represents the point of coordinates $(0,0), \ldots, (n-1,0)$, respectively, by using the formulae

$$\begin{aligned}
P_{0,0}(i) &:= \bigwedge_{k=0}^{n} D_{q_k(i)} \wedge \bigwedge_{k=0}^{n} D_{p_k(i)} \\
P_{1,0}(i) &:= \bigwedge_{k=0}^{n} D_{q_k(i)} \wedge \neg D_{p_0}(i) \wedge \bigwedge_{k=1}^{n} D_{p_k(i)} \\
P_{2,0}(i) &:= \bigwedge_{k=0}^{n} D_{q_k(i)} \wedge D_{p_0}(i) \wedge \neg D_{p_1}(i) \wedge \bigwedge_{k=2}^{n} D_{p_k(i)} \\
&\ldots
\end{aligned}$$

The existence of a tiling is then expressed by the satisfiability of the formula below (the first conjunct takes care of the initialization, whereas the last two about tile matching):

$$\begin{aligned}
&\bigwedge_{k=0}^{n-1} \forall i \, (P_{k,0}(i) \rightarrow a[i] = d_k) \wedge \\
&\wedge \ \forall i_1 \forall i_2 \, (H_I(i_1, i_2) \rightarrow H_E(a[i_1], a[i_2])) \wedge \\
&\wedge \ \forall i_1 \forall i_2 \, (V_I(i_1, i_2) \rightarrow V_E(a[i_1], a[i_2])) \quad .
\end{aligned}$$

Notice that the above (polynomially long) formula is in the $\forall^*$-monic-flat fragment, as it can be seen by inspecting the definitions of the macros we used for for $P_{k,0}(i), V_I(i_1, i_2), H_I(i_1, i_2)$. □

Lemma 6.3.4 (Uppper Bound)**.** $\mathtt{ARR}^2(\mathcal{LIA}, \mathcal{LIA})$*-satisfiability of* $\exists^*\forall^*$*-monic-flat sentences is in* NExpTime.

Proof. To show the matching upper bound, it is sufficient to inspect our decision algorithm $\mathsf{SAT}_{\mathsf{MULTI}}$. Clearly, Step I introduces an exponential guess; the formulæ F_1, F_2, F_3, F_I, F_E are all exponentially long (notice that there are exponentially many σ in F_1 and $\mathfrak{B}$ can be seen as a set of exponentially many literals). It is well-known that $\mathcal{LIA}$-satisfiability of quantifier-free formulæ is in NP (see the historical references in [Oppen, 1978] for the origins of this result), so that satisfiability of F_E also takes non deterministic exponential time. We only have to discuss $\mathcal{LIA}$-satisfiability of F_I in more detail. Now, F_I is not quantifier-free and in order to check its satisfiability we need to run a quantifier

elimination procedure to the subformula

$$\neg\,\exists x\,\neg\,\left(\bigvee_{j=1}^{q}\bigwedge_{L\in M_j} L(x,\mathbf{c})\right) \tag{6.12}$$

The point is that this formula is exponentially long and so we must carefully analyze the cost of the elimination of a single existential quantifier in $\mathcal{LIA}$. We need the following lemma from [Oppen, 1978] (Theorem 1, p.327):

Lemma 6.3.5. *Suppose that Cooper's quantifier elimination algorithm, applied to a formula $\exists x\,\phi$ (with quantifier-free ϕ) yields the quantifier-free formula ϕ'. Let c_0 (resp. c_1) be the number of distinct positive integers appearing as indexes of divisibility predicates or as variable coefficients within ϕ (resp. ϕ'); let s_0 (resp. s_1) be the largest absolute values of integer constants (including coefficients) occurring in ϕ (resp. ϕ'); let a_0 (resp. a_1) be the number of atoms of ϕ (resp. ϕ'). Then the following relationship hold:*

$$c_1 \leq c_0^4, \qquad s_1 \leq s_0^{4c_0}, \qquad a_1 \leq a_0^4 s_0^{2c_0}.$$

Now notice that (6.12) is exponentially long, but integer constants, integer coefficients and indexes of divisibility predicates are the same as in the input formula. Thus, if N bounds the length of the input formula, we get a $2^{O(N^2)}$-bound for the above parameters c_1, s_1, a_1 for the formula ϕ' resulting from the elimination of the universal quantifier from (6.12). Now (quoting from [Oppen, 1978], p.329), "the space required to store [a formula] F_k is bounded by the product of the number of atoms a_k in F_k, the maximum number $m+1$ of constants per atom, the maximum amount of space s_k required to store each constant, and some constant q (included for the various arithmetic and logical operators, etc.)." This means that our ϕ' is exponentially long and, as a consequence, our satisfiability testing for F_I works in NExPtime, as it applies an NP algorithm to an exponential instance. □

6.4 Related work

The modular nature of our solution makes our contributions orthogonal with respect to the state of the art: we can enrich $\mathcal{LIA}$ with various definable or even not definable symbols [Semënov, 1984] and get from our Theorems 6.2.1,6.3.1 decidable classes which are far from the scope of existing results. Given the parameterized nature of our results, there is some similarity with [Ihlemann

et al., 2008], although (contrary to [Ihlemann et al., 2008]) we consider purely syntactically specified classes of formulæ. It is interesting to notice that also the special cases of the decidable classes covered by Corollary 6.2.1 and Theorem 6.3.2 are orthogonal to the results from the literature. To this aim, we make a closer comparison with [Habermehl et al., 2008a, Bradley et al., 2006].

The two fragments considered in [Habermehl et al., 2008a, Bradley et al., 2006] are characterized by rather restrictive syntactic constraints. In [Habermehl et al., 2008a] it is considered a subclass of the $\exists^*\forall$-fragment of $\mathtt{ARR}^1(T)$ called SIL, *Single Index Logic*. In this class, formulæ are built according to a grammar allowing (i) as atoms occurring in universally quantified subformulæ only difference logic constraints and some equations modulo a fixed integer and (ii) as universally quantified subformulæ only formulæ of the kind $\forall \mathbf{i}.\phi(\mathbf{i}) \rightarrow \psi(\mathbf{i}, \mathbf{a}(\mathbf{i}+\mathbf{k}))$ (here $\mathbf{k}$ is a tuple of integers) where ϕ, ψ are conjunctions of atoms (in particular, no disjunction is allowed in ψ). On the other side, SIL includes some non-flat formulæ, due to the presence of constant increment terms $\mathbf{i}+\mathbf{k}$ in the consequents of the above universally quantified implications. Similar restrictions are in [Habermehl et al., 2008b].

The Array Property Fragment described in [Bradley et al., 2006] is basically a subclass of the $\exists^*\forall^*$-fragment of $\mathtt{ARR}^2(\mathcal{LIA}, \mathcal{LIA})$; however universally quantified subformulæ are constrained to be of the kind $\forall \mathbf{i}.\phi(\mathbf{i}) \rightarrow \psi(\mathbf{a}(\mathbf{i}))$, where in addition the `INDEX` part $\phi(\mathbf{i})$ is restricted to be a conjunction of atoms of the following four kinds: $i \leq j, i \leq t, t \leq i$ (with $i, j \in \mathbf{i}$ and where t does not contain occurrences of the universally quantified variables $\mathbf{i}$). These formulæ are flat; they may not be monic because of the atoms $i \leq j$.

From a computational point of view, a complexity bound for $\mathsf{SAT}_{\mathsf{MONO}}$ has been shown in the proof of Theorem 6.2.1, while the complexity of the decision procedure proposed in [Habermehl et al., 2008a] is unknown. On the other side, both $\mathsf{SAT}_{\mathsf{MULTI}}$ and the decision procedure described in [Bradley et al., 2006] run in NExpTime. The decision procedure in [Bradley et al., 2006] is in NP only if the number of universally quantified index variables is bounded by a constant N (this is not the case of $\mathsf{SAT}_{\mathsf{MULTI}}$, where with two universally quantified index variables the NExpTime lower and upper bounds are attained).

Our decision procedures for quantified formulæ are also partially different, in spirit, from those presented so far in the SMT community. While the decidability of the vast majority of SMT-Solvers address the problem of checking the satisfiability of quantified formulæ via instantiation (see, e.g., [Bradley et al., 2006, Detlefs et al., 2003, Ge and de Moura, 2009, Reynolds et al., 2013]), our procedure $\mathsf{SAT}_{\mathsf{MULTI}}$ is still based on instantiation, but the instantiation

refers to a set of terms enlarged with the free constants witnessing the guessed set of realized types. Notice also that $\mathsf{SAT}_{\mathsf{MULTI}}$ introduces in Step II (see section 6.3.1) a universally quantified arithmetic subformula to be handled in Step V (for the lack of a better method) via quantifier-elimination; a similar remark applies also to $\mathsf{SAT}_{\mathsf{MONO}}$, thus the generation of quantified purely arithmetic sub-goals is an additional specific feature of our satisfiability procedures.

De Moura and Bjørner and Goel et al. presented in [de Moura and Bjørner, 2009] and [Goel et al., 2008], respectively, two interesting work describing a decision procedure for the theory of arrays proposed by McCarthy in [McCarthy, 1962], i.e., the theory having as a signature the symbols $\{\_[\_], \mathsf{store}(\_, \_, \_)\}$ which interpretation is constrained by the two axioms

$$\forall a, i, v.\mathsf{store}(a, i, v)[i] = v$$
$$\forall a, i, j, v.(i = j \vee \mathsf{store}(a, i, v)[j] = a[j])$$

These works are interested for us since some Flat Array Properties can, in fact, be expressed without quantifiers by exploiting the signature symbols of the theory of arrays and handled by the two ground decision procedures for arrays presented in the aforementioned works. We point out that, however, enlarging the signature of a theory is a double-edged sword. If, on the one hand, expressiveness is augmented, on the other hand (the SMT side) the decision procedure has to be enhanced to deal with the newly introduced symbols.

6.5 Summary

In this chapter we presented a new decidable subfragment of the $\exists^*\forall^*$-fragment of the theories of arrays. We called it *Flat Array Properties*. Indeed flatness, along with some other monic constraints, is the key for stating our new decidability results. Our new decision procedures are parameterized in terms of the theories describing the indexes and elements of the arrays. Required features for the decidability of Flat Array Properties decidability are met by theories widely used in practice, e.g., Linear Arithmetic.

We studied Flat Array Properties of the mono-sorted theory of arrays (section 6.2) and the multi-sorted theory of arrays (section 6.3). For the former case we provided a general complexity analysis of our decision procedure, parameterized in the complexity of the "base theory" to which free function symbols have been added to model arrays. For the latter case, we studied the complexity for deciding the satisfiability of (monic) Flat Array Properties of the multi-sorted

theory of arrays $\mathtt{ARR}^2(\mathcal{LIA}, \mathcal{LIA})$, showing that this is a NExpTime-complete problem.

We point out that in [Alberti et al., 2014c, Alberti et al., 2015] we gave experimental evidence that the class of Flat Array Properties admits formulæ that do require our ad-hoc decision procedure for checking their decidability, given that no available solver (up to the publishing date of [Alberti et al., 2014c, Alberti et al., 2015]) is able to detect their (un)satisfiability. This show that the fragment of Flat Array Property is not included into known decidable fragments of the theories of arrays [Bradley et al., 2006, Habermehl et al., 2008b, Ge and de Moura, 2009].

6.5.1 Related publications

The results reported in this chapter have been published in the following papers:

- F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for Flat Array Properties. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*. Springer, 2014.

- F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for Flat Array Properties. *Journal of Automated Reasoning*, 54(4):327–352.

Chapter 7

Deciding the safety of a class of programs with arrays

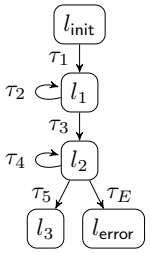
In this chapter we show that the safety of an interesting class of programs handling arrays or strings of unknown length is decidable. We call this class of programs $\mathsf{simple}^0_{\mathcal{A}}$-*programs*: this class covers non-recursive programs implementing for instance searching, copying, comparing, initializing, replacing and testing functions. The method we use to show these safety results is similar to a classical method adopted in the model-checking literature for programs manipulating integer variables (see for instance [Bozga et al., 2009c, Comon and Jurski, 1998, Finkel and Leroux, 2002]): we first assume flatness conditions on the control flow graph of the program and then we assume that transitions labeling cycles are "acceleratable". The key point is that the shape of most accelerated transitions from [Alberti et al., 2013b] matches the definition of Flat Array Properties. This fact with some constraints over the control-flow structure of the programs allow to design an acceleration-based decision procedure, generating finitely many Flat Array Properties which unsatisfiability determines the safety of a given $\mathsf{simple}^0_{\mathcal{A}}$-program.

7.1 Background

This chapter builds on the results of chapters 5 and 6. We therefore assume the notions introduced in sections 5.1 and 6.1 along with the more general background notions of chapter 2. In particular, as a reference theory, we shall use $\mathtt{ARR}^1(\mathcal{LIA}^+)$ or $\mathtt{ARR}^2(\mathcal{LIA}^+, \mathcal{LIA}^+)$, where $\mathcal{LIA}^+$ is $\mathcal{LIA}$ enriched with free constant symbols and with *definable* predicate and function symbols. Recall from section 2.2.3 that definable symbols are nothing but useful macros that can

```
procedure initEven ( a[N] , v ) {
    for (i = 0; i < N; i = i + 2)  a[i] = v;
    for (i = 0; i < N; i = i + 2)  assert(a[i] = v);
}
```

(a)



(b)

Figure 7.1. The initEven procedure (a) and its control-flow graph (b).

be used to formalize case-defined functions. Below, we let $\mathcal{T}$ be $\mathtt{ARR}^1(\mathcal{LIA}^+)$ or $\mathtt{ARR}^2(\mathcal{LIA}^+, \mathcal{LIA}^+)$.

In this chapter it is convenient to consider the following definition of programs:

Definition 7.1.1 (Programs)**.** *Given a set of variables* $\mathbf{v}$*, a* program *is a triple* $\mathbf{P} = (L, \Lambda, E)$*, where (i)* $L = \{l_1, \ldots, l_n\}$ *is a set of* program locations *among which we distinguish, as usual, an initial location* l_{init} *and an error location* l_{error}*; (ii)* Λ *is a finite set of transition formulæ* $\{\tau_1(\mathbf{v}, \mathbf{v}'), \ldots, \tau_r(\mathbf{v}, \mathbf{v}')\}$ *and (iii)* $E \subseteq L \times \Lambda \times L$ *is a set of* actions*.*

Obviously, any array-based transition system defined as a quadruple $(\mathbf{v}, l_{\text{init}}, l_{\text{error}}, T)$ (see section 2.3 and Definition 2.3.2) induces a program (L, Λ, E) matching Definition 7.1.1. We also assume the availability of the three projection function on E indicated by $src, \mathcal{L}, trg$, that is, for $e = (l_i, \tau_j, l_k) \in E$, we have $src(e) = l_i$, $\mathcal{L}(e) = \tau_j$ (this is called the 'label' of e) and $trg(e) = l_k$.

Example 7.1.1. Consider the procedure initEven, taken from [Dillig et al., 2010], in Figure 7.1. For this procedure, $\mathbf{a} = a$, $\mathbf{s} = i, v$. N is a constant of the background theory. Λ is the set of formulæ (we omit identical updates):

$$
\begin{aligned}
\tau_1 &:= i' = 0 \\
\tau_2 &:= i < N \wedge a' = \lambda j.\mathsf{if}\ (j = i)\ \mathsf{then}\ v\ \mathsf{else}\ a(j) \wedge i' = i + 2 \\
\tau_3 &:= i \geq N \wedge i' = 0 \\
\tau_4 &:= i < N \wedge a(i) = v \wedge i' = i + 2 \\
\tau_5 &:= i \geq N \\
\tau_E &:= i < N \wedge a(i) \neq v
\end{aligned}
$$

The procedure initEven can be formalized as the control-flow graph depicted in Figure 7.1(b), where $L = \{l_{\text{init}}, l_1, l_2, l_3, l_{\text{error}}\}$.

Definition 7.1.2 (Program paths)**.** *A* program path *(in short,* path*) of* $\mathbf{P} = (L, \Lambda, E)$ *is a sequence* $\rho \in E^n$*, i.e.,* $\rho = e_1, e_2, \ldots, e_n$*, such that for every* e_i, e_{i+1}*,* $trg(e_i) = src(e_{i+1})$*. We denote with* $|\rho|$ *the length of the path. An* error path *is a path* ρ *with* $src(e_1) = l_{\mathsf{init}}$ *and* $trg(e_{|\rho|}) = l_{\mathsf{error}}$*. A path* ρ *is a* feasible path *if* $\bigwedge_{j=1}^{|\rho|} \mathcal{L}(e_j)^{(j)}$ *is* $\mathcal{T}$*-satisfiable, where* $\mathcal{L}(e_j)^{(j)}$ *represents* $\tau_{i_j}(\mathbf{v}^{(j-1)}, \mathbf{v}^{(j)})$*, with* $\mathcal{L}(e_j) = \tau_{i_j}$ *(the notation* $\tau_{i_j}(\mathbf{v}^{(j-1)}, \mathbf{v}^{(j)})$ *means that we made copies* $\mathbf{v}^{(j-1)}, \mathbf{v}^{(j)}$ *of the program variables* $\mathbf{v}$ *and we replaced* $\mathbf{v}, \mathbf{v}'$ *by them in* $\tau(\mathbf{v}, \mathbf{v}')$*).*

The *(unbounded) reachability problem* for a program $\mathbf{P}$ is to detect if $\mathbf{P}$ admits a feasible error path. Proving the safety of $\mathbf{P}$, therefore, means solving the reachability problem for $\mathbf{P}$. Notably, this definition is equivalent to the Definition 2.4.1 stating what does it mean to be *safe* for an array-based transition system.

7.2 A decidability result for the reachability analysis of flat array programs

To gain decidability, we must first impose restrictions on the shape of the transition formulæ, for instance we can constrain the analysis to formulæ falling within decidable classes like those we analyzed in chapter 6. This is not sufficient however, due to the presence of loops in the control flow. Hence we assume flatness conditions on such control flow and "accelerability" of the transitions labeling self-loops. This is similar to what is done in [Bozga et al., 2009c, Comon and Jurski, 1998, Finkel and Leroux, 2002] for integer variable programs, but since we handle array variables we need specific restrictions for acceleration.

We first give the definition of flat^0-program, i.e., programs with only self-loops for which each location belongs to at most one loop. Subsequently we will identify sufficient conditions for achieving the full decidability of the reachability problem for flat^0-programs.

Definition 7.2.1 (flat^0-program)**.** *A program* $\mathbf{P}$ *is a* flat^0-program *if for every path* $\rho = e_1, \ldots, e_n$ *of* $\mathbf{P}$ *it holds that for every* $j < k$ *(*$j, k \in \{1, \ldots, n\}$*), if* $src(e_j) = trg(e_k)$ *then* $e_j = e_{j+1} = \cdots = e_k$*.*

We now turn our attention to transition formulæ. Recall that, given a loop represented as a transition relation τ, the accelerated transition τ^+ allows to compute *in one shot* the *precise* set of states reachable after n unwindings

of that loop, for any n. This prevents divergence of the reachability analysis along τ, caused by its unwinding. As discussed in chapter 5, an obstacle for the applicability of acceleration in the domain we are targeting is that accelerations are not always definable in the logical formalisms we consider. Based on this observation, on definability of accelerations, we are now ready to state a general result about the decidability of the reachability problem for programs with arrays. The theorem we give is modular and general. We will show instances of this result in the next sections. Notationally, let us extend the projection function $\mathcal{L}$ by putting $\mathcal{L}^+(e) := \mathcal{L}(e)^+$ if $src(e) = trg(e)$ and $\mathcal{L}^+(e) := \mathcal{L}(e)$ otherwise, where $\mathcal{L}(e)^+$ denotes the acceleration of the transition labeling the edge e.

Theorem 7.2.1. *Let $\mathcal{F}$ be a class of formulæ decidable for $\mathcal{T}$-satisfiability. The unbounded reachability problem for a flat^0-program $\mathbf{P}$ is decidable if*

(i) $\mathcal{F}$ is closed under conjunctions and

(ii) for each $e \in E$ one can compute $\alpha(\mathbf{v}, \mathbf{v}') \in \mathcal{F}$ such that $\mathcal{T} \models \mathcal{L}^+(e) \leftrightarrow \alpha(\mathbf{v}, \mathbf{v}')$.

Proof. Let $\rho = e_1, \ldots, e_n$ be an error path of $\mathbf{P}$; when testing its feasibility, according to Definition 7.2.1, we can limit ourselves to the case in which $e_1, \ldots, e_n$ are all distinct, provided we replace the labels $\mathcal{L}(e_k)^{(k)}$ with $\mathcal{L}^+(e_k)^{(k)}$ in the formula $\bigwedge_{j=1}^{n} \mathcal{L}(e_j)^{(j)}$ from Definition 7.1.2.[1] Thus $\mathbf{P}$ is unsafe iff, for some path $e_1, \ldots, e_n$ whose edges are all distinct, the formula

$$\mathcal{L}^+(e_1)^{(1)} \wedge \cdots \wedge \mathcal{L}^+(e_n)^{(n)} \tag{7.1}$$

is $\mathcal{T}$-satisfiable. Since the involved paths are finitely many and $\mathcal{T}$-satisfiability of formulæ like (7.1) is decidable, the safety of $\mathbf{P}$ can be decided. □

7.3 A class of array programs with decidable reachability problem

We are now ready to identify a class of programs with arrays – we call it $\mathsf{simple}^0_{\mathcal{A}}$-programs– for which requirements of Theorem 7.2.1 are met. The class of $\mathsf{simple}^0_{\mathcal{A}}$-programs contains non recursive programs implementing searching,

[1] Notice that by these replacements we can represent in one shot infinitely many paths, namely those executing self-loops any given number of times.

copying, comparing, initializing, replacing and testing procedures. As an example, the initEven program reported in Figure 7.1 is a $\mathsf{simple}^0_{\mathcal{A}}$-program. In order to formalize the notion of $\mathsf{simple}^0_{\mathcal{A}}$-program we need the notion of simple_k-assignments. Simple_k-assignments are transitions (defined below) for which the acceleration is first-order definable and is a Flat Array Property. For an integer number k, we denote by k the term $1 + \cdots + 1$ (k-times) and by $k \cdot t$ the term $t + \cdots + t$ (k-times).

Definition 7.3.1 (simple_k-assignment)**.** *Let $k \neq 0$; a simple_k-assignment is a transition $\tau(\mathbf{v}, \mathbf{v}')$ of the kind*

$$\phi_L(\mathbf{s}, \mathbf{a}(d)) \wedge d' = d + k \wedge \mathbf{d}' = \mathbf{d} \wedge \mathbf{a}' = \lambda j.\mathsf{if}\ (j = d)\ \mathsf{then}\ \mathbf{t}(\mathbf{s}, \mathbf{a}(d))\ \mathsf{else}\ \mathbf{a}(j)$$

where (i) $\mathbf{s} = d, \mathbf{d}$ and (ii) the formula $\phi_L(\mathbf{s}, \mathbf{a}(d))$ and the terms $\mathbf{t}(\mathbf{s}, \mathbf{a}(d))$ are flat.

Definition 7.3.2 ($\mathsf{simple}^0_{\mathcal{A}}$-programs)**.** *A $\mathsf{simple}^0_{\mathcal{A}}$-program $\mathbf{P} = (L, \Lambda, E)$ is a flat^0-program such that (i) every $\tau \in \Lambda$ is a formula belonging to one of the decidable classes covered by Corollary 6.2.1 or Theorem 6.3.2; (ii) if $e \in E$ is a self-loop, then $\mathcal{L}(e)$ is a simple_k-assignment.*

To understand the above notation, recall that according to our conventions, if $\mathbf{a} = a_1, \ldots, a_s$, then $\mathbf{a}(d)$ means the s-tuple of terms $a_1(d), \ldots, a_s(d)$; moreover, $\mathbf{t}(\mathbf{s}, \mathbf{a}(d))$ stands for an s-tuple of terms $t_1(\mathbf{s}, \mathbf{a}(d)), \ldots, t_s(\mathbf{s}, \mathbf{a}(d))$. Finally, $\mathbf{a}' = \lambda j(\cdots)$ stands for a conjunction of s-equations updating the tuple $\mathbf{a}$, where the $\lambda j(\cdots)$ notation indicates the s-tuple of functions which are defined by the displayed macros. The formula $\mathbf{a}' = \lambda j(\cdots)$ can thus be rewritten as a plain first order formula as follows

$$\bigwedge_{h=1}^{s} \forall j. \left(\begin{array}{c} (j = d \wedge a'_h(j) = t_h(\mathbf{s}, \mathbf{a}(d))) \vee \\ \vee\ (j \neq d \wedge a'_h(j) = a_h(j)) \end{array} \right) \tag{7.2}$$

In a simple_k-assignment, the arrays $\mathbf{a}$ are scanned by the counter d, the cells $\mathbf{a}(d)$ are overwritten and the counter is then increased by k. It would be possible to extend the definition and the upcoming result to transitions requiring different scanners for the different arrays (one scanner for each of them) with different increments. In order to not complicating further the notation we prefer to skip this easy generalization.

The following Lemma is an instance of the Theorem 5.2.2 and gives the template for the accelerated counterpart of a simple_k-assignment.

Lemma 7.3.1. *Let $\tau(\mathbf{v}, \mathbf{v}')$ be a simple_k-assignment like in Definition 7.3.1. Then $\tau^+(\mathbf{v}, \mathbf{v}')$ is $\mathcal{T}$-equivalent to the formula*

$$\exists y > 0 \begin{pmatrix} \forall z.\, (d \leq z < d + k \cdot y \wedge D_k(z - d) \rightarrow \ \phi_L(z, \mathbf{d}, \mathbf{a}(d))) \ \wedge \\ \mathbf{a}' = \lambda j.\mathbf{U}(j, y, \mathbf{v}) \ \wedge \ d' = d + k \cdot y \ \wedge \ \mathbf{d}' = \mathbf{d} \end{pmatrix} \tag{7.3}$$

where the definable functions $U_h(j, y, \mathbf{v})$, $1 \leq h \leq |\mathbf{a}|$, of the tuple of functions $\mathbf{U}$ are

$$\mathsf{if}\ (d \leq j < d + k \cdot y \ \wedge D_k(j - d))\ \mathsf{then}\ b_h(j, \mathbf{d}, \mathbf{a}(j))\ \mathsf{else}\ a_h(j)\ .$$

Proof. It is sufficient to check by induction on $y \geq 1$ that is we execute y-times the simple_k-assignment of Definition 7.3.1, we get

$$\begin{aligned} &\forall z.\, (d \leq z < d + k \cdot y \wedge D_k(z - d) \rightarrow \ \phi_L(z, \mathbf{d}, \mathbf{a}(d))) \ \wedge \\ &\wedge \ \mathbf{a}' = \lambda j.\mathbf{U}(j, y, \mathbf{v}) \ \wedge \ d' = d + k \cdot y \ \wedge \ \mathbf{d}' = \mathbf{d} \end{aligned}$$

which means that the accelerated assignment is described by (7.3). □

Example 7.3.1. Consider transition τ_2 from the formalization of our running example of Figure 7.1. The acceleration τ_2^+ of such formula is (we omit identical updates)

$$\exists y > 0.\, \begin{pmatrix} \forall z.(i \leq z < i + 2y \wedge D_2(z - i) \rightarrow z < N) \wedge i' = i + 2y \ \wedge \\ a' = \lambda j.\, (\mathsf{if}\ \ (i \leq j < 2y + i \wedge D_2(j - i))\ \ \mathsf{then}\ v\ \mathsf{else}\ a(j)) \end{pmatrix}$$

We can now formally show that the reachability problem for $\mathsf{simple}^0_{\mathcal{A}}$-programs is decidable, by instantiating Theorem 7.2.1 with the results obtained so far.

Theorem 7.3.1. *The unbounded reachability problem for $\mathsf{simple}^0_{\mathcal{A}}$-programs is decidable.*

Proof. By prenex transformations, distributions of universal quantifiers over conjunctions, etc., it is easy to see that the decidable classes covered by Corollary 6.2.1 or Theorem 6.3.2 are closed under conjunctions. Since the acceleration of a simple_k-assignment fits inside these classes (just eliminate definitions via λ-abstractions by using universal quantifiers, like in (7.2)), Theorem 7.2.1 applies. □

7.4 Summary

This chapter presented decidability results for establishing the safety of programs handling arrays. In general, the problem of checking the safety of programs is undecidable, given its relation to the halting problem [Turing, 1936].

The constraints on which we built our result characterize both the control-flow structure of the program and the shape of the relations declaratively encoding the instructions of the program. We showed that decidability of the safety analysis can established for programs with flat control-flow structure and for which all the loops admit an acceleration falling in a decidable fragment (section 7.2).

We subsequently instantiated this general result in the case of programs handling arrays, exploiting the results presented in chapter 5 and chapter 6 (section 7.3).

7.4.1 Related publications

The results reported in this chapter have been published in the following papers:

- F. Alberti, S. Ghilardi, and N. Sharygina. Acceleration-based safety decision procedure for programs with arrays. In K. L. McMillan, A. Middeldorp, G. Sutcliffe, and A. Voronkov, editors, *LPAR 2013, 19th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, December 12-17, 2013, Stellenbosch, South Africa, Short papers proceedings*, volume 26 of *EPiC Series*, pages 1–8. EasyChair, 2013.

- F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for Flat Array Properties. In E. Ábrahám and K. Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*. Springer, 2014.

- F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for Flat Array Properties. *Journal of Automated Reasoning*, 54(4):327–352.

Chapter 8

Booster: a verification framework for programs with arrays

This last chapter of the thesis presents Booster, a framework for the verification of programs with arrays. The main feature of Booster, differentiating it with respect to other tools offering a support for the analysis of programs with arrays (e.g., [Bjørner et al., 2013, Cousot et al., 2011, Hoder et al., 2011, De Angelis et al., 2014b, Garg et al., 2014, Dragan and Kovács, 2014]), it is being based on a framework integrating quite standard abstraction-based solutions with innovative acceleration procedures.

As stated in the introduction, this combination can be achieved thanks to the fact that we work on a declarative level. The fact that all the techniques we presented in this thesis work on formulæ allows for the design of a framework combining all of them. The intuition behind the integrated framework implemented in Booster is that acceleration and abstraction have orthogonal strengths and weaknesses. A combined framework will take the best from such techniques overcoming their individual limitations. With respect to abstraction-based procedures, acceleration offers a *precise* solution (not involving over-approximations) to the problem to compute the reachable state-space of a transition system, but on the other side has syntactic restrictions preventing its general application. On the other side, abstraction-based solutions are usually a very general framework, but they also require heuristics (and in some cases even user guidance) in order to increase their practical effectiveness.

Beside offering an implementation of the techniques presented in the thesis, Booster includes a front-end for a C-like programming language, some preprocessing techniques and internal heuristics selecting the best options (e.g., term abstraction list) for the execution of its analysis techniques. This makes

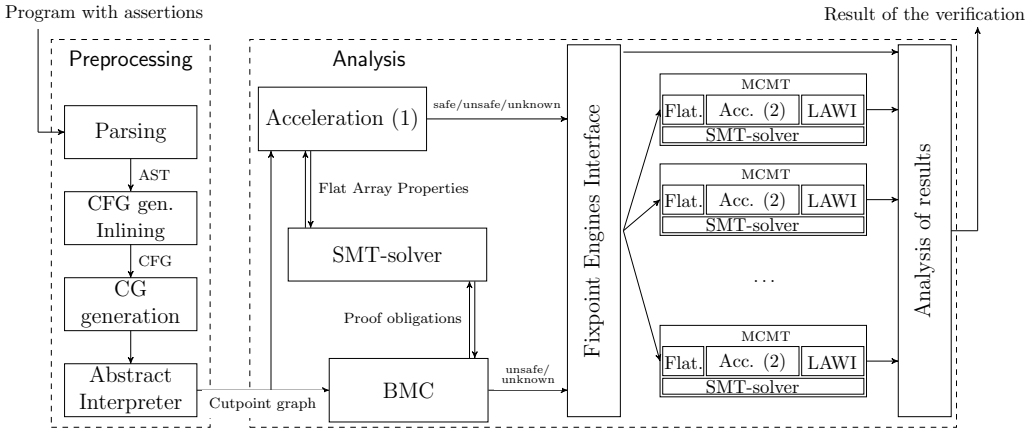


Figure 8.1. The architecture of BOOSTER.

the verification of programs *completely automatic*.

BOOSTER is structured according to the standard compilers architecture: the initial parsing phase generates an intermediate representation of the code which is subject to several optimizations before being fed to different modules implementing some formal analysis technique for checking its safety.

The architecture of the tool will be described in the next section. This chapter discusses also an extensive experimental evaluation of BOOSTER on a large class of examples taken from different heterogeneous sources (section 8.2). This benchmark suite includes all the examples of other suites of relevant related work, e.g., the examples from [Dillig et al., 2010]. To the best of our knowledge, BOOSTER is the *only* tool able to deal with the set of examples in our benchmark suite. Notably, the relevance of the BOOSTER benchmark suite for the software model-checking research community is witnessed by the fact that a large part of them became part of the SV-COMP (Software Verification Competition, `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/c/array-examples/`) starting from the 19th of September, 2014[1].

8.1 Architecture of BOOSTER

Figure 8.1 depicts the architecture of BOOSTER. The features of the tool are described in the following subsections.

[1]BOOSTER accepts a formalisms for quantified assertions. The benchmarks with quantified assertions are not part of the SV-COMP.

8.1.1 Preprocessing

BOOSTER parses a C-like language. It accepts `int` and `bool` scalar variables, arrays of `int` and `bool`. Program can have multiple procedures in addition to the `main` one, but not recursive procedures. We allow *quantified assertions*, which are convenient for writing in a readable and compact way interesting properties over arrays. These assertions are recognized by the grammar

$$\texttt{assert}\ (\ \texttt{forall}\ (\mathit{vars\_decl})\ ::\ \mathit{bool\_expr})$$

where *vars_decl* and *bool_expr* matches respectively a valid `C` declaration of a sequence of scalar (integer) variables and a `C` Boolean expression. We also assume that all the arrays have an unknown, unbounded length, and the pass-by-reference paradigm when array variables are passed as arguments to the methods of the program. Non-initialized variables (or array cells) are not implicitly assigned to a default value. This implies that if BOOSTER verifies a program, the program is safe for *any* value of the uninitialized variables and array cells.

Given a program, BOOSTER generates its control-flow graph (CFG) and inlines procedure calls. Each block of the CFG contains a sequence of instructions which can be only assignments or assumptions. The CFG generated by BOOSTER has one entry block, the natural starting point of the program, and two exit blocks: one reachable by all the executions of the program correctly terminating and the other one reachable by those executions violating some assertions in the code. From the CFG, BOOSTER builds the *cutpoint graph* (CG) of the input program [Gurfinkel et al., 2011]. A cutpoint graph is a graph-representation of the input code where each vertex represents either the entry/exit block of the program or a loop-head, and the edges are labeled with sequences of assumptions or assignments. The representation of the input code as a cutpoint graph is adopted to maximize the application of acceleration procedures. Indeed, acceleration techniques can be applied only to transitions representing self-loops (and matching some other syntactic patterns, as discussed in chapter 5 and chapter 7). Consider the pseudocode in Figure 8.2(a). The naïve CFG representation of the loop, reported in Figure 8.2(b), involves a first edge linking the *while* condition and the *if-then-else* condition. Two other edges represent the two branches and a fourth edge goes back from a "join location" after the *if-then-else* to the loop head. Translating such code representation into a transition system prevents the application of acceleration procedures since no transition will represent a self-loop. A cutpoint graph rep-

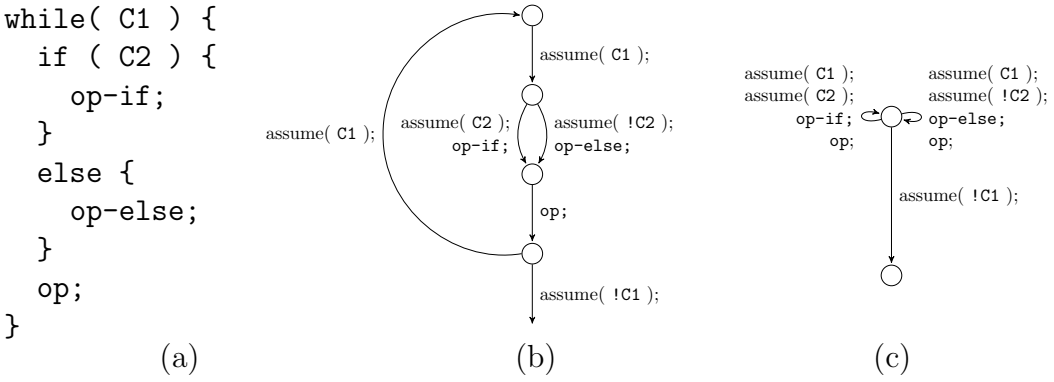


Figure 8.2. Two equivalent representations of the same program. The one on the right allows for the application of acceleration procedures.

resentation of the same code has, however, only two self-loops over the same vertex (Figure 8.2(c)). Both loops will be analyzed by the acceleration procedure.

8.1.2 Abstract Interpreter

Abstract interpretation has been considered, so far, as one of the most efficient approaches for inferring inductive invariants of programs. Abstract interpretation targets the efficient (i.e., at compile time) generation of properties in some abstract domain of interest. An abstract domain can be thought as a (fragment of a) theory [Gulwani and Tiwari, 2006]. The main differences between abstract interpretation and the solutions presented so far in this thesis is represented by the fact that the inductive invariants produced by an abstract interpreter are not ensured to be safe. Abstract interpretation solutions target efficiency with the counter-effect of returning false alarms, usually due to the application of join or widening operators (the latter one, being required for ensuring convergence of the technique).

Since the seminal Cousots' paper [Cousot and Cousot, 1977], many different abstract domains have been studied. An abstract domain identifies the properties one can infer. For example, the *interval domain* [Cousot and Cousot, 1977] is not powerful enough, in general, to check desired comparison between two scalar variables in the code, as it targets the generation of invariants of the kind $c_1 \leq x \leq c_2$ for a variable x and two numerical constants $c_1, c_2 \in [-\infty, +\infty]$. The *polyhedral abstraction* is more precise, as it allows the inference of linear relationship between scalar program variables. The price of higher precision of this domain is the highest computational complexity, which implies an un-

avoidable loss of efficiency. The abstract domain *octagons* constitutes a fairly good compromise between the interval domain and the polyhedral domain, as it targets the generation of relations of the kind $\pm x + \pm y \leq c$. On the other side, its has cubic complexity in time[2]. Logozzo and Fähndrich present in [Logozzo and Fähndrich, 2010] another abstract domain useful for inferring relations over pairs of variables which is less precise but at the same time cheaper from a computational point of view than the *octagon* domain. It is the *pentagon* abstract domain. It allows to infer properties of the kind $c_1 \leq x \leq c_2 \wedge x < y$, for pairs of variables x, y and rationals c_1, c_2.

In this context, abstract interpretation can be of great help for two main reasons. First, the abstract interpreter we implemented generates invariants by exploring the program in a *forward* fashion while our fixpoint engine works backwardly. Mixing forward and backward analyses has been shown to be a winning technique in many cases (see, e.g., [De Angelis et al., 2014a]). Second, BOOSTER implements an abstract interpreter working on convex polyhedra [Cousot and Halbwachs, 1978, Bagnara et al., 2003]. This is the abstract domain $\mathcal{P} = (\mathbb{P}, \sqsubseteq, \sqcup, \sqcap, \nabla)$ where $\mathbb{P}$ is the infinite set of all possible linear inequalities over the scalar variables $\mathbf{s}$ of the program $\mathcal{S}_\mathcal{T}$, $\sqsubseteq$ is a partial order over $\mathbb{P}$, $\sqcup$ and $\sqcap$ are respectively the join and the meet operators of the lattice $(\mathbb{P}, \sqsubseteq)$ and ∇ is a widening operator. We assume that our abstract interpreter computes a standard upward Kleene iteration sequence over $\mathcal{P}$ driven by program instructions over the scalars. Operations on arrays are treated as follows: array reads return undefined values, array writes are ignored. Convergence to a fixpoint is guaranteed by the application of the widening operator ∇, as defined in [Bagnara et al., 2003]. The abstract interpreter takes, therefore, as input an array-based transition system $\mathcal{S}_\mathcal{T}$ and returns for each control location an *inductive invariant*, that is, an element of $\mathbb{P}$ closed by post-image computation with respect to the τ's of $\mathcal{S}_\mathcal{T}$, which therefore includes all reachable states at that location. After converting, for each program location l, the invariant into a first-order formula $C_l(\mathbf{v})$, we obtain an inductive invariant for $\mathcal{S}_\mathcal{T}$, satisfied by all reachable states:

$$K(\mathbf{s}) \;:=\; \bigwedge_{l \in L} pc = l \rightarrow C_l(\mathbf{s})$$

where $C_l(\mathbf{s})$ is a linear inequality with integer coefficients over the program scalar variables $\mathbf{s}$.

Notably, it is rather hard, in general, to infer the facts $C_l(\mathbf{s})$ as interpolants,

[2] A precise analysis is given by Bagnara et al. in [Bagnara et al., 2005].

even with the enhancing of interpolation procedures with several heuristics for tuning the quality of the interpolants. This approach, on the contrary, produces them almost *for free*. In our case, term abstraction can leverage such additional lemmas to discover new unsat cores by abstracting away more terms, resulting in the generation of more general interpolants.

8.1.3 Acceleration (1)

This module targets the verification of $\mathsf{simple}^0_{\mathcal{A}}$-programs, as defined in section 7.3. From Definition 7.3.2, these are programs characterized by (i) having a flat control-flow structure, i.e., each location belongs to at most one loop, and (ii) comprising only loops that can be accelerated as Flat Array Properties. If the given CG is a $\mathsf{simple}^0_{\mathcal{A}}$-program, BOOSTER accelerates all the loops. This is a cheap template-based pattern matching task. The loops are substituted with their accelerated counterparts; subsequently BOOSTER generates the proof-obligations, which are Flat Array Properties, required to check the (un)safety of the program. Unfortunately, this fragment is not entirely covered by decision procedures implemented in available SMT-solvers. In practice, BOOSTER relies on the Z3 SMT-solver [de Moura and Bjørner, 2008] for solving such queries. The SMT-solver is usually very efficient on unsatisfiable proof obligations, but might struggle on satisfiable ones (an example is given later). The BMC analysis executed before this module, however, is generally able to find the corresponding traces, reporting the unsafety of the code before starting this acceleration procedure. It is important to notice that, at this stage of the analysis, BOOSTER exploits the *full power* of acceleration on a well-defined class of transitions, i.e., the loops of $\mathsf{simple}^0_{\mathcal{A}}$-programs. Conversely, the technique implemented in the "Acceleration (2)" module inside the fixpoint engine MCMT (described later), applies to a wider class of transitions but intractable formulæ generated by the acceleration are over-approximated.

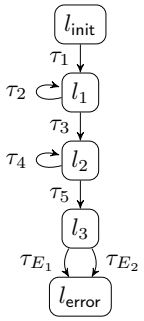
A concrete example of Flat Array Property

As an example of Flat Array Property where Z3 fails[3], consider the mergeInterleave procedure, taken from [Dillig et al., 2010] and reported in Figure 8.3(a). The formal representation of this procedure according to the definitions of chapter 6 and chapter 7 is the following: $\mathbf{a} = a, b, r$, $\mathbf{c} = i, k$. N is a constant of the background theory. Λ is the set of formulæ (we omit identical updates

[3]To the best of our knowledge, this formula is out of reach for all the available SMT-solvers.

```
procedure mergeInterleave ( a[N] , b[N] , r[N] , k ) {
    for (i = 0; i < N; i = i + 2)  r[i] = a[i];
    for (i = 1; i < N; i = i + 2)  r[i] = b[i];
    if(0 ≤ k ∧ k < N ∧ k ≡2 0) assert(r[k] = b[k]);
    if(0 ≤ k ∧ k < N ∧ k ≡2 1) assert(r[k] = a[k]);
}
```

(a)



(b)

Figure 8.3. The mergeInterleave procedure (a) and its control-flow graph (b).

and the transitions not leading to error locations):

$$\begin{aligned}
\tau_1 &:= i' = 0 \\
\tau_2 &:= i < N \wedge r' = \lambda j.\text{if } (j = i) \text{ then } a(j) \text{ else } r(j) \wedge i' = i + 2 \\
\tau_3 &:= i \geq N \wedge i' = 1 \\
\tau_4 &:= i < N \wedge r' = \lambda j.\text{if } (j = i) \text{ then } b(j) \text{ else } r(j) \wedge i' = i + 2 \\
\tau_5 &:= i \geq N \\
\tau_{E_1} &:= k \geq 0 \wedge k < N \wedge k \equiv_2 0 \wedge r[k] \neq b[k] \\
\tau_{E_2} &:= k \geq 0 \wedge k < N \wedge k \equiv_2 1 \wedge r[k] \neq a[k]
\end{aligned}$$

The procedure mergeInterleave can be formalized as the control-flow graph depicted in Figure 8.3(b) (as before, we are not reporting edges of the control-flow graph that are not considered for checking the safety of the procedure), where $L = \{l_{\text{init}}, l_1, l_2, l_3, l_{\text{error}}\}$.

Transitions τ_2 and τ_4 are simple_k-assignments. Their accelerations are (omitting identical updates):

$$\tau_2^+ := \exists y. \begin{pmatrix} y > 0 \wedge i' = i + 2y \wedge \\ \forall j.((i \leq j < i + 2y \wedge D_2(j - i)) \rightarrow j < N) \wedge \\ r' = \lambda j.\text{if } (i \leq j < 2y + i \wedge D_2(j - i)) \text{ then } a(j) \text{ else } r(j) \end{pmatrix}$$

and

$$\tau_4^+ := \exists y. \begin{pmatrix} y > 0 \wedge i' = i + 2y \wedge \\ \forall j.((i \leq j < i + 2y \wedge D_2(j - i)) \rightarrow j < N) \wedge \\ r' = \lambda j.\text{if } (i \leq j < 2y + i \wedge D_2(j - i)) \text{ then } b(j) \text{ else } r(j) \end{pmatrix}$$

The procedure mergeInterleave is not safe: a possible execution run showing the unsafety is $\tau_1 \wedge \tau_2^+ \wedge \tau_3 \wedge \tau_4^+ \wedge \tau_5 \wedge \tau_{E_1}$, because r is initialized in the even positions with elements from a, not from b. The error trace is the Flat Array Property:

$$
\begin{array}{l}
i_1 = 0 \wedge \forall j.r_1(j) = r_0(j) \wedge \\
\exists y_1. \left(\begin{array}{l} y_1 > 0 \wedge i_2 = i_1 + 2y_1 \wedge \\ \forall j.((i_1 \leq j < i_1 + 2y_1 \wedge D_2(j - i_1)) \rightarrow j < N) \wedge \\ \forall j.(r_2(j) = \mathsf{if}\ (i_1 \leq j < 2y_1 + i_1 \wedge D_2(j - i_1))\ \mathsf{then}\ a(j)\ \mathsf{else}\ r_1(j)) \end{array} \right) \wedge \\
i_2 \geq N \wedge i_3 = 1 \wedge \forall j.(r_3(j) = r_2(j)) \wedge \\
\exists y_3. \left(\begin{array}{l} y_3 > 0 \wedge i_4 = i_3 + 2y_3 \wedge \\ \forall j.((i_3 \leq j < i_3 + 2y_3 \wedge D_2(j - i_3)) \rightarrow j < N) \wedge \\ \forall j.(r_4(j) = \mathsf{if}\ (i_3 \leq j < 2y_3 + i_3 \wedge D_2(j - i_3))\ \mathsf{then}\ b(j)\ \mathsf{else}\ r_3(j)) \end{array} \right) \wedge \\
i_4 \geq N \wedge i_5 = i_4 \wedge \forall j.(r_5(j) = r_4(j)) \wedge \\
0 \leq k \wedge k < N \wedge D_2(k) \wedge r_5(k) \neq b(k) \wedge \\
i_6 = i_5 \wedge \forall j.(r_6(j) = r_5(j))
\end{array}
$$

This formula is, to the best of our knowledge, not solvable by any available SMT-solver.

8.1.4 Bounded Model Checking

As we introduced at the very beginning of this thesis, *Bounded Model Checking* (BMC) is a technique introduced two decades ago in the arena of software model-checking techniques [Biere et al., 1999]. Given a transition system $\mathcal{S}_\mathcal{T} = (\mathbf{v}, l_{\mathsf{init}}, l_{\mathsf{error}}, T)$, this techniques generates the formulæ

$$pc^{(n)} = l_{\mathsf{init}} \ \wedge \ \bigwedge_{i=1}^{n} T(\mathbf{v}^{(i)}, \mathbf{v}^{(i-1)}) \ \wedge \ pc^{(0)} = l_{\mathsf{error}} \tag{2.6}$$

for all n up to a given N, and checks their $\mathcal{T}$-satisfiability. The technique is inherently incomplete, meaning that it can only prove the unsafety of $\mathcal{S}_\mathcal{T}$ and can do that only if $\mathcal{S}_\mathcal{T}$ admits a counterexample of length $m \leq N$. Recall from Theorem 3.3.1 that the $\mathcal{T}$-(un)satisfiability of (2.6) is decidable in our case.

The role played by BMC inside Booster is to detect the unsafety of programs *before* enabling analysis (like acceleration) with a high impact on the tool performance. Indeed, formulæ generated by the "Acceleration (1)" are

Flat Array Properties, and we have proven in section 6.3.3 that checking their satisfiability may be a NEXPTIME-complete problem. A low number of unwindings constitutes, at this stage of the analysis, a good trade-off between precision (number of unsafe programs detected) and efficiency.

8.1.5 Transition System generation

If the program is not a $\mathsf{simple}^0_{\mathcal{A}}$-program or the SMT-solver exploited by the "Acceleration (1)" module times out, the CG of the program is translated into a transition system and then fed into the fixpoint engine.

8.1.6 Fixpoint engine – MCMT

The fixpoint engine included in BOOSTER is an enhanced version of MCMT, where approaches of chapter 3 with the heuristics of chapter 4 and the solutions of chapter 5 have been combined. MCMT performs three main operations. It applies a flattening procedure to the input transition system, accelerates all the transitions it can accelerate and then executes the LAWI approach. When a spurious counterexample arises it checks whether it contains an accelerated transition or not. In the former case it applies the refinement procedure described in section 5.3.2, in the latter the interpolation-based refinement of section 3.3. The choice of MCMT with respect to SAFARI is to overcome the intrinsic limitations of OPENSMT, in particular because of its not offering decision procedures for $\mathcal{LIA}$.

Flattening

Flattening is a preprocessing technique exploited to reduce the transition formulæ and state formulæ to a flat format, i.e., where array variables are indexed only by existentially quantified variables (recall Definition 2.2.1). It exploits the rewriting rule $\phi(a[t], ...) \rightsquigarrow \exists x(x = t \wedge \phi(a[x], ...))$. Recall from chapter 4 that this format is particularly indicated for inferring *quantified* predicates within the LAWI framework and it is exploited by the *term abstraction* heuristic.

Acceleration (2)

MCMT adopts acceleration as a preprocessing step, following the approach described in chapter 5. In contrast with the "Acceleration (1)" module discussed

previously, acceleration here is applied to a wider class of transitions, but preimages along accelerated formulæ are not kept precise given their intractable format and are over-approximated with their *monotonic abstraction* as discussed in section 5.3 by performing finite instantiations of the universal quantifiers over a set of terms $\mathcal{I}$ automatically retrieved from the formula itself.

Lazy Abstraction With Interpolants

This module implements the approach discussed in chapter 3, inheriting all the heuristics presented in chapter 4. The term abstraction list is automatically generated by BOOSTER. BOOSTER generates term abstraction lists containing symbolic constants and iterators of the program.

8.1.7 Portfolio approach

As shown in chapter 4, the Term Abstraction heuristic has a great impact on the effectiveness of the LAWI framework for arrays. One of the biggest limitations of Term Abstraction, however, is its requiring a term abstraction list to select the terms to abstract away while generating the interpolants.

BOOSTER nullifies the required user ingenuity for defining a proper term abstraction list. Internal heuristics, inherited from SAFARI, generate some suitable term abstraction lists. The fixpoint engine is subsequently executed adopting a portfolio approach, according to which BOOSTER generates several parallel instances of MCMT, each with different settings (including different term abstraction lists).

8.2 Experimental evaluation

We evaluated BOOSTER on more than 200 programs (both safe and unsafe) with arrays taken from the following sources:

- Programs where an array is exploited to implement a set. We verify that the inserting and deleting procedures maintains the property stating that the array does not contain duplicates.
- `www.sanfoundry.com/c-programming-examples-arrays/`. Some of the program on the web-page are not interesting from our point of view (e.g., some of them are there only to show how to print array elements, for teaching purposes, and do not exhibit any interesting array manipulation algorithm).

- Well-known sorting procedures.
- Relevant literature on solutions for the analysis of array programs. Some of these programs have been deliberately modified in order to test the strength of the tool. For example, the "copy$N$.c" programs have N consecutive loops, each copying one array into a new one[4]. Notably, for a CEGAR-based analysis these examples are rather challenging, as counterexamples will go through several loops and each loop may be unwound different number of times within the same counterexample.
- SV-COMP repository, "loops" folder.

All experiments have been executed on a computer equipped with an Intel(R) Xeon(R) CPU @ 2.40GHz and 16GB of RAM. Booster was executed with the following parameters:

- Z3 timeout: 500 ms
- bmc depth: 1
- parallel executions of MCMT: 40
- number of iterations before applying widening: 3

Running time has been measured always with the `time` utility, taking the first result (flagged as "real").

In this section we want to evaluate the performance of Booster depending on the different combinations of techniques enabled. As discussed before, the analysis techniques implemented in Booster are abstract interpretation, acceleration (precise, for programs admitting a decidable reachability analysis), acceleration (approximated) and abstraction[5]. In all the graphics that we will show in this chapter, these techniques will be represented by the acronyms AI, DP, Acc and Abs. The standard precise backward reachability is represented by a -. Each graphic will plot the value of bivariate variables having as coordinates the running time of Booster executed with two different settings. Information about which techniques have been evaluated in each graph is given by the labels of the axes. For example, the label "AI Acc" states that the values of each point in the plot for that axis is the running time of Booster

[4]For example, "copy3.c" copies an input array a into a new array b, b into a new array c, c into a new array d. The property we check in the end is that a is equal to d.

[5]BMC is enabled only when the precise acceleration procedure runs.

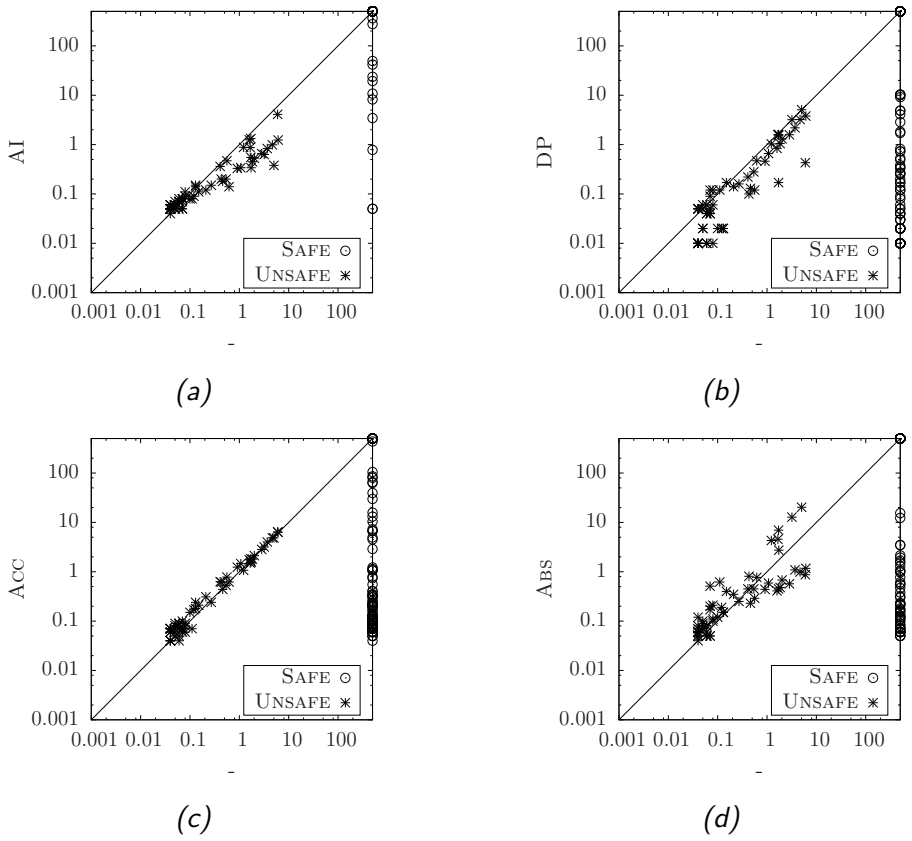


Figure 8.4. Comparison between the precise backward reachability procedure and the precise backward reachability procedure enhanced with abstract interpretation (a), precise acceleration (b), approximated acceleration (c), lazy abstraction with interpolants (d).

executed on one example of our benchmark suite enabling abstract interpretation and the approximated version of acceleration, leaving disabled both the precise acceleration procedure and the abstraction feature. We display the diagonal for each plot. This line helps in showing which techniques "win" on which benchmark: a point below the diagonal indicates that the setting of the y-axis performs better than the setting on the x-axis, and vice-versa.

8.2.1 Advantages over precise backward reachability

We want to evaluate, at first, the contribution of each technique implemented in Booster to limiting divergence of the standard precise backward reachability analysis. Graphics in Figure 8.4 compare the backward reachability analysis

with and without the enhancing of the static analysis techniques implemented in BOOSTER, that is Abstract Interpretation, Acceleration (both approximated and precise) and lazy abstraction with interpolants.

As expected, backward reachability is not able to detect the safety of the input code for any of the safe problems, while detects the unsafety of all the programs with a bug.

The graphics show also that all the techniques contribute in limiting divergence. Not surprisingly, the two most-effective techniques are the approximated acceleration and lazy abstraction with interpolants. The precise acceleration procedure succeeds only on those examples matching a strict templates (see chapter 6). In addition, the abstract interpretation module succeeds on a few examples. These are benchmarks where, despite the presence of arrays, the required invariant is a property over the scalars of the code. For such programs it is sufficient to generate a quantifier-free safe inductive invariant.

From this evaluation we can also draw another conclusion: none of the techniques slows down the tool on the entire set of unsafe benchmarks. All the graphics in Figure 8.4 indicate that the different techniques can achieve a speed-up on some benchmarks and be slower with respect to the precise backward reachability on some other benchmarks.

8.2.2 Benefits of each technique

We now want to evaluate the benefits of each technique when compared with the others. That is, we evaluate the benefits of a single technique against the benefits given by the other techniques alone and all together. The general outcome, as we shall discuss in the following sections, is that *all* the techniques we implemented in BOOSTER offer some advantages making them mandatory for some benchmarks where they succeed while all the other techniques fail. This points out that an integrated framework like BOOSTER has higher chances of success, compared with those tools based on one, single analysis technique. We now discuss the benefits of each technique in details.

Abstract Interpretation

The abstract domain implemented inside BOOSTER works with the polyhedra abstract domain. It cannot infer quantified properties. As the graphics in Figure 8.5 show, BOOSTER generally fails on safe instances if only the abstract interpreter is enabled. However, all the plots except the one in Figure 8.5d admit some circle on the above y-axis, meaning that there are examples for

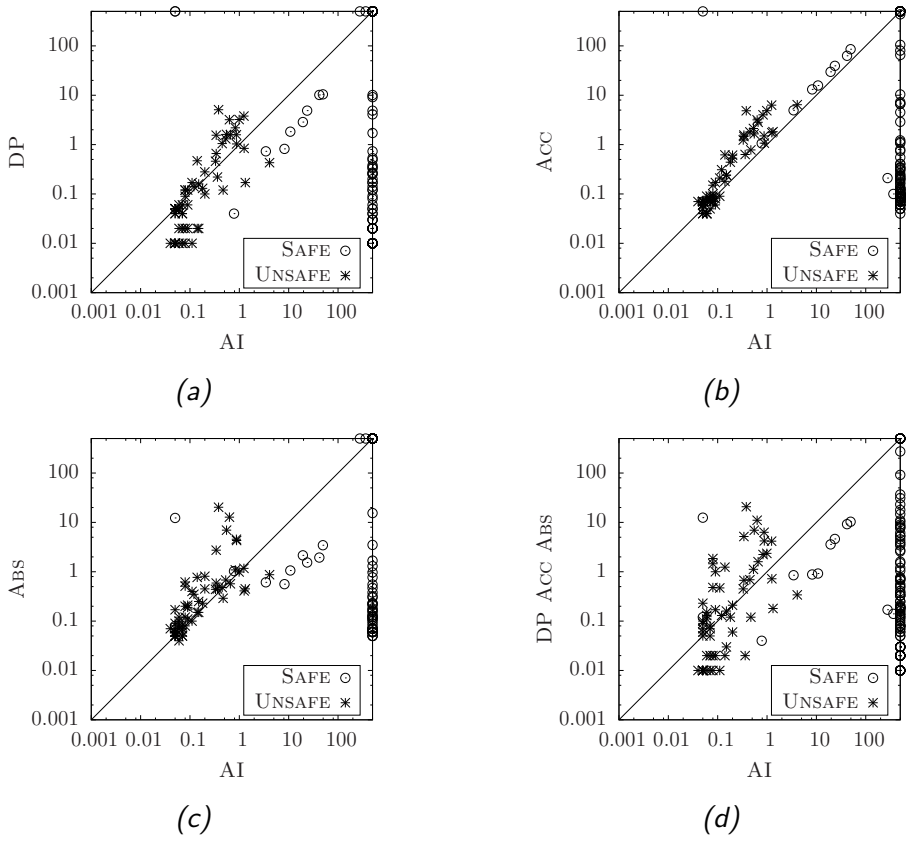


Figure 8.5. Strength of abstract interpretation with respect to precise acceleration (a), approximated acceleration (b), lazy abstraction with interpolants (c) and the three techniques together (d).

which the abstract interpreter wins against the other techniques, if applied alone. Such examples, however can be solved if acceleration, both precise and approximated, and lazy abstraction with interpolants are enabled. Still, however, there are examples on the up-right corner of Figure 8.5d. This means that there are examples for which neither the abstract interpreter alone nor the other three techniques combined can solve. These examples can be actually solved, as we shall discuss later, by combining the four techniques together.

Precise acceleration

Precise acceleration has a clear and precise target, i.e., the benchmarks which cutpoint graph is a $\mathsf{simple}^0_{\mathcal{A}}$-program. For such programs it is possible to produce a finite number of formulæ such that their satisfiability implies the

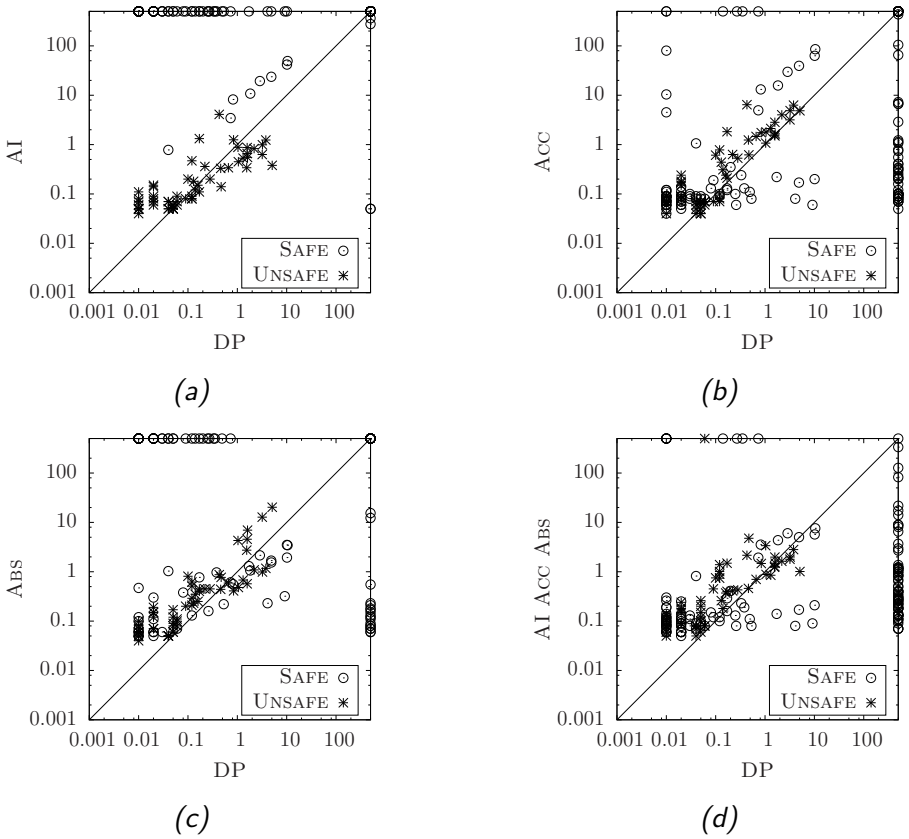


Figure 8.6. Strength of *precise acceleration* with respect to abstract interpretation (a), approximated acceleration (b), lazy abstraction with interpolants (c) and the three techniques together (d).

presence of a bug in the code and their unsatisfiability indicates a valid safe inductive invariant proving the safety of the program. To overcome possible slow-downs due to the fact that our decision procedure is not currently implemented in any SMT-solver[6], we enable in parallel a BMC module in charge of detecting unsafe programs. Given the architecture of BOOSTER, if the input code is not a $\mathsf{simple}^0_{\mathcal{A}}$-program, BOOSTER will execute only a precise backward reachability on the code. The outcome, for such examples is the one discussed earlier[7]. The benefits of this module is that it guarantees the termination of the analysis on a class of benchmarks on which the other techniques (alone or combined) fail. Indeed, all the graphics of Figure 8.6 admits circles on the

[6]In this case, the solver might timeout returning `unknown` instead of `sat` or `unsat`.

[7]BMC is enabled only for $\mathsf{simple}^0_{\mathcal{A}}$-programs.

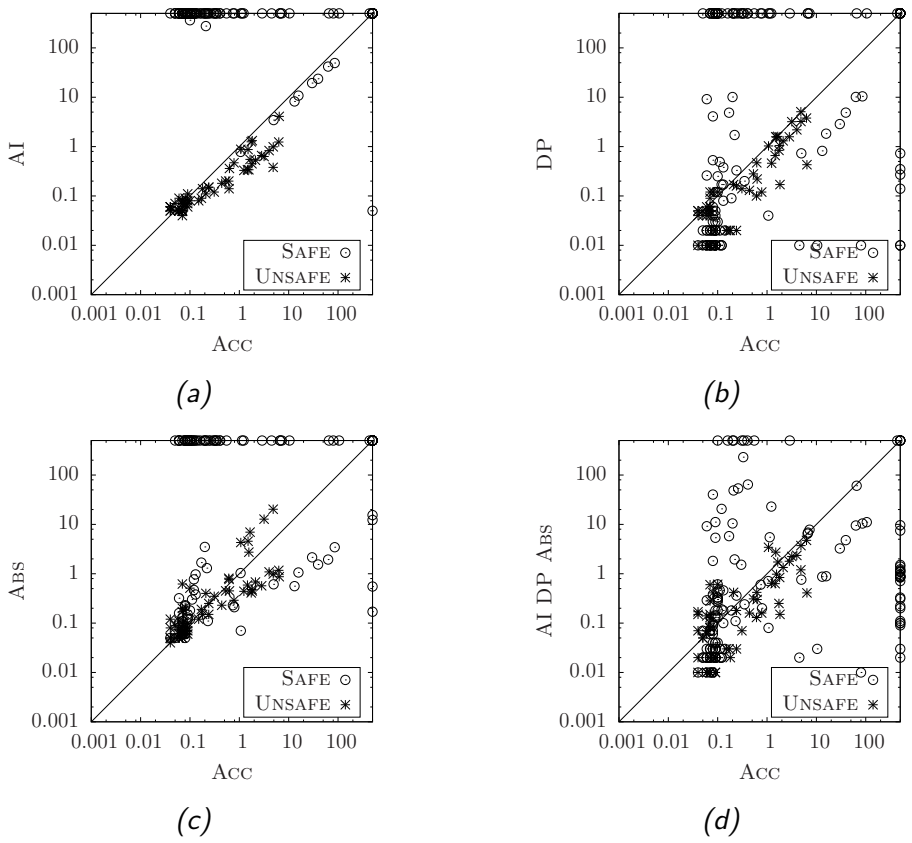


Figure 8.7. Strength of *approximated acceleration* with respect to abstract interpretation (a), precise acceleration (b), lazy abstraction with interpolants (c) and the three techniques together (d).

above x-axis. For some safe programs, this technique is also faster than the others.

Approximated acceleration

Approximated acceleration compromises precision for effectiveness. As discussed in chapter 5, this technique generates formulæ that might be outside of decidable fragments of the theories of arrays we assume as the background theory. For this reason, it applies an approximation step to make the technique effective in practice.

The graphics in Figure 8.7 report the evaluation of the strength of the approximated acceleration module in comparison with the other techniques implemented in Booster. From Figure 8.7a it is clear that this technique is

superior, on safe instances, to the single abstract interpreter, even though the right y-axis reports a few circles, representing programs for which this analysis times out, at the contrary of the configurations where the abstract interpreter is enabled.

Figure 8.7b witnesses the fact that the two accelerations have different strengths and weaknesses: both of them fail on some programs (safe and unsafe) and perform better on different set of benchmarks.

Figure 8.7c shows that this module and the abstraction module have orthogonal strengths and weaknesses, witnessed by the fact that they timeout on different safe examples. The unsafe benchmarks follow the same pattern of the safe ones, albeit they are not reporting divergence but only different performance.

In the end, Figure 8.7d shows that acceleration alone brings an important advantage inside Booster: a relevant portion of the benchmarks are represented by circles or crosses above the diagonal. For this examples, the approximated acceleration module enables an analysis which is more effective than the other three combined.

Lazy Abstraction with Interpolants

Lazy Abstraction with Interpolants is the last module we discuss in this section. It is the last analysis that can be enabled in Booster, running at the end of the architectural pipeline of the model-checker.

This analysis technique targets generality: as opposed to acceleration procedures, where strict syntactic patterns determine whether the technique can be applied or not, acceleration operates on every array-based transition system. Its chances of success depend on different heuristics, though, as discussed in chapter 4. Here we notice that the graphics of Figure 8.8 confirm our previous findings. Abstraction is useful given its generality, but have to be coupled with other techniques that, different in spirit, are able to limit its divergence.

8.2.3 Acceleration vs. Abstraction

Another important evaluation is the one measuring the benefits of abstraction (in its two different shapes) and acceleration (where precise and approximated versions are combined). The graphic is shown in Figure 8.9. From this graphic it is clear that abstraction and acceleration should be combined in order to achieve important results. Indeed, both techniques fail on different portions of our benchmark suite. Their combination, as it has been implemented in

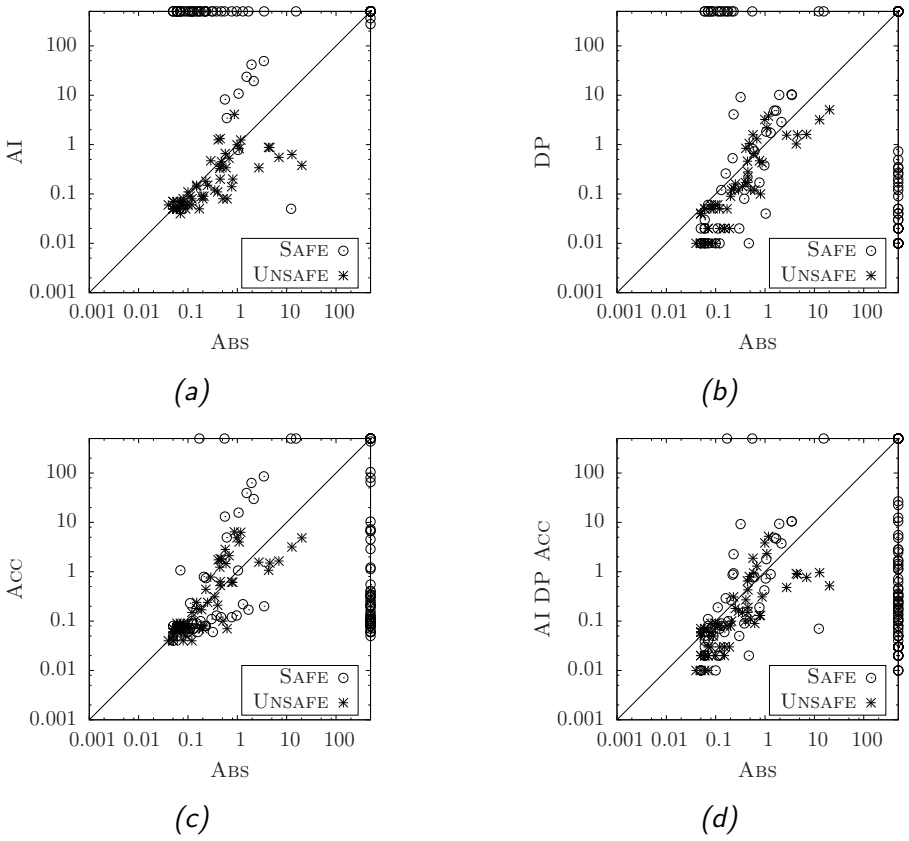


Figure 8.8. Strength of *lazy abstraction with interpolants* with respect to abstract interpretation (a), precise acceleration (b), approximated acceleration (c) and the three techniques together (d).

Booster, leads to a success on both sets where only one fails. In addition, the combination of the two techniques succeeds also on examples where both techniques fail. This is the case, for example, of programs with nested loops. Acceleration alone fails because it can deal only with the inner loop. Abstraction, on the other side, is not able to generate a safe inductive invariant because the counterexamples traverse the two loops. This means that the chain of interpolants for one counterexample has to contain facts that will contribute to two different loop invariants. On the other side, the combination of the two analysis will be beneficial since acceleration will avoid divergence along the inner loop, and abstraction will have to deal with only the external one.

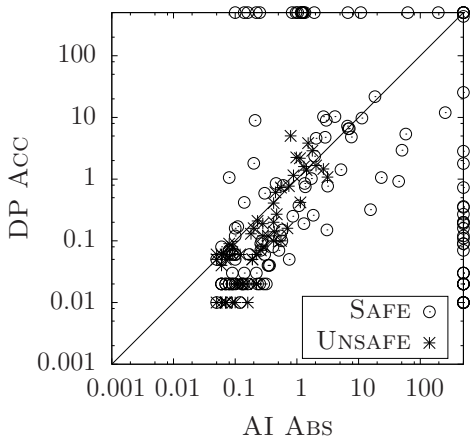


Figure 8.9. Comparison between abstraction and acceleration.

8.2.4 The combined framework

As a final experimental comparison, we report in Figure 8.10 the graphics showing the performance of Booster with all the techniques enabled and Booster with all but one the techniques enabled. It is clear from the graphics that enabling all the techniques allow to achieve the best results. It is not excluded that there might be little slowdowns: Figure 8.10d, for example, compares Booster with all the techniques enabled and Booster with only abstraction disabled. Some safe programs are reported below the diagonal. This means that for those benchmarks, abstraction is not necessary and introduces an overhead slowing down the tool. However, in none of the four graphics of Figure 8.10 there are circles on the right y-axes. This shows that the combination of the four techniques is able to solve benchmarks that are out of reach for Booster with some techniques disabled.

8.3 Summary

In this chapter we presented a framework, called Booster, for the verification of programs with arrays. It is written in C++, and it is available at `http://verify.inf.usi.ch/booster`. The framework builds upon the results presented in the entire thesis, and combines them efficiently in order to overcome their individual limitations. The declarative playground is the key for being able to integrate all such techniques and enrich them with other state-of-the-art static analysis solutions (in the case of Booster, abstract in-

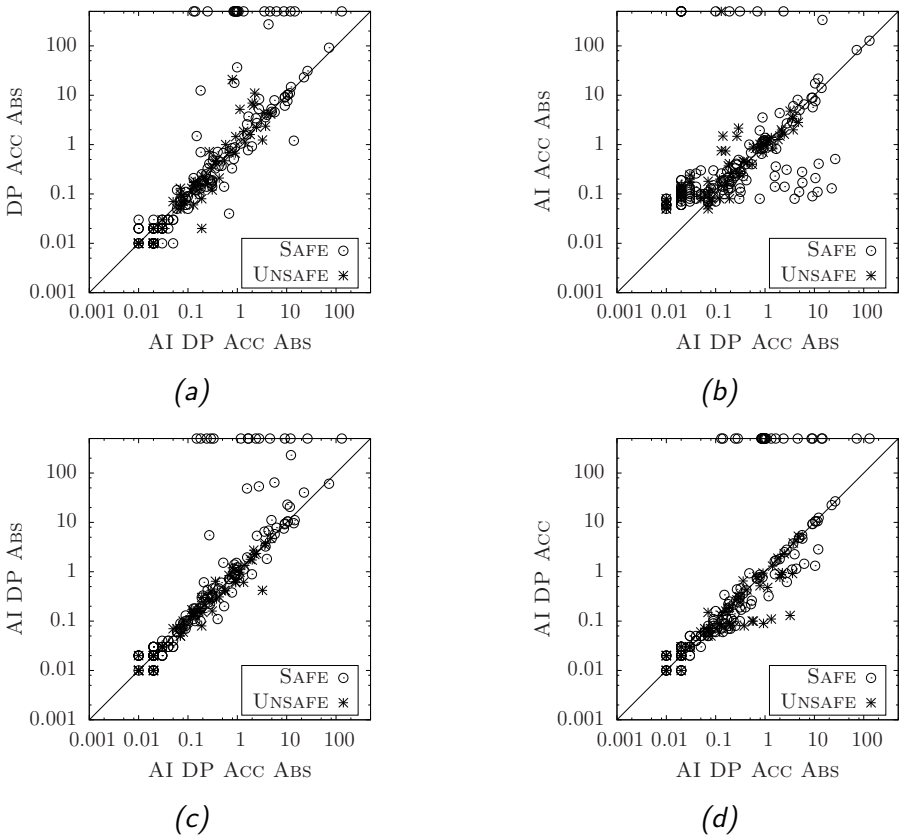


Figure 8.10. Strength of Booster with all its features enabled with respect to Booster when one of its feature is disabled: abstract interpretation (a), precise acceleration (b), approximated acceleration (c) and the lazy abstraction with interpolants (d).

terpretation and bounded model-checking).

An extensive experimental analysis concludes the chapter. The experiments empirically show the effectiveness of Booster on a wide set of state-of-the-art benchmarks. In addition, we discussed the benefit of every singular analysis implemented in Booster providing a detailed discussion about the benefits of acceleration over abstraction and vice-versa.

8.3.1 Related publications

The results reported in this chapter have been published in the following papers:

- F. Alberti, S. Ghilardi, and N. Sharygina. A framework for the verification of parameterized infinite-state systems. In L. Giordano, V. Gliozzi,

and G.L. Pozzato, editors, *Proceedings of the 29th Italian Conference on Computational Logic, Torino, Italy, June 16-18, 2014*, volume 1195 of *CEUR Workshop Proceedings*, pages 303–308. CEUR-WS.org, 2014.

- F. Alberti, S. Ghilardi, and N. Sharygina. Booster: An acceleration-based verification framework for array programs. In F. Cassez and J.-F. Raskin, editors, *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, volume 8837 of *Lecture Notes in Computer Science*, pages 18–23. Springer, 2014.

- F. Alberti, and D. Monniaux Polyhedra to the rescue of array interpolants. In *SAC 2015*. To appear.

Chapter 9

Conclusions

Efficient and effective solutions for the analysis of programs with arrays require the ability to generate and handle quantified formulæ representing meaningful properties about their executions. The need for quantifiers invalidates the vast majority of existing frameworks for the static analysis of software systems and presents new challenges.

In this thesis we addressed the problem of verifying programs with arrays from different points of view, taking into considerations both "high-level" theoretical problems, like studying the definability of the acceleration of a class of relations used to declaratively encode pieces of programs, to "low-level" pragmatic issues, like implementing instantiation procedures for developing sound static analysis tools.

The first contribution of the thesis is given in chapter 3. We presented a new verification framework for the analysis of programs with arrays following the "Lazy Abstraction with Interpolant" approach [McMillan, 2006], where refinement is performed by computing interpolants from unsatisfiable formulæ encoding spurious counterexamples. The generation of quantified safe inductive invariants is achieved by pipelining a preprocessing procedure, introducing quantified variables in the transition relation, and a refinement one generating quantifier-free interpolants to refine the explored state-space. In section 3.3 we identified a fragment of the theory of arrays admitting quantifier-free interpolation. Section 3.4 targets the study of suitable hypothesis for the termination of the backward (CEGAR-based) reachability analysis we presented.

Chapter 4 discusses an efficient implementation of the results of chapter 3. The issues we target in chapter 4 arise from the needs of handling quantified formulæ and tuning the refinement procedure. The former problem has been tackled by devising practical (and necessary incomplete) instantiation proce-

dures and strategies for exploring the reachable state-space in a convenient way delaying the introduction of new quantified variables. The latter problem, instead, has been mitigated with the help of two heuristics, "term abstraction" and "counterexample minimization". Term abstraction tunes interpolation procedures in order to compute more general interpolants increasing the chances to generate a safe inductive invariant. Counterexample minimization limits the changes to the explored state-space preferring local and peripheral refinements to global and more invasive ones. Experimentally these two techniques pay off and allow to achieve good experimental results.

As a future direction in this field, it is unavoidable that the framework we presented in chapters 3 will be "contaminated" by the IC3 philosophy. Started with the work of Aaron Bradley, [Bradley, 2011], IC3 (or PDR, Property Directed Reachability) received a lot of attention in the last two years. After the initial improvements proposed on the propositional level [Eén et al., 2011], IC3 has been lifted to the first-order level by many authors, e.g., [Cimatti and Griggio, 2012, Hoder and Bjørner, 2012]. The most recent advantages in this research thread are, to the best of our knowledge, those presented by Cimatti et al. in [Cimatti et al., 2014] and Birgmeier et al. in [Birgmeier et al., 2014]. There are no IC3-based tools that are able to work with parameterized systems. It would be interesting, therefore to pursue this direction. All the papers presenting re-implementation of old algorithms and ideas in a new IC3-like style show an improvement of the performances. On the other side, IC3 offers only a new way for exploring the state-space and reacting to the detection of too coarse abstraction, the so called "counterexample to induction", but does not target the generation of more general interpolants or other problems lying at the core of abstraction-based solution. The techniques presented in chapter 4 will be therefore likely needed as well in new IC3-based solutions for the verification of parameterized systems.

In our application domain, two issues have to be solved in order to develop a verification framework. One need to keep under control both the introduction of quantifiers and the quality of the interpolants. The standard theory of arrays does not allow quantifier-free interpolation [Kapur et al., 2006] and such theoretical limitation might prevent the effectiveness of static analyzer relying on interpolation theorem provers like iZ3 [McMillan, 2011], given the unpredictable shape of interpolants. In this setting, a general strategy for achieving quantifier-free interpolation is by enlarging the signature of a theory. This has been done for the linear arithmetic case, over $\mathbb{Z}$: The divisibility predicates are required to establish quantifier-free interpolation. The works [Totla and Wies, 2013, Bruttomesso et al., 2012b] apply this idea to the theory of arrays.

Enlarging the signature has a drawback, though. If the new signature becomes unmanageable from a computational point of view, the gain obtained with quantifier-free interpolation is lost. The second issue limiting the efficiency of interpolation-based refinement is the randomness of interpolants. The term abstraction heuristic, discussed in section 4.1.1, allows to compute better interpolants. Term abstraction has been generalized in [Rümmer and Subotic, 2013]. Other works addressing the same goal of Term Abstraction are mostly on the propositional level [Rollini et al., 2013]. It would be interesting to understand the effect of such techniques on a first-order logic level.

Another contribution of the thesis is the definition of an acceleration framework for the analysis of programs with arrays. In chapter 5 we addressed the problem of divergence of backward reachability from a different perspective with respect to the one taken in chapter 3. Inspired by the works on integer variables [Bozga et al., 2009a, Bozga et al., 2009b, Bozga et al., 2010], we studied whether acceleration would have been applicable in the context of the analysis of programs with arrays. Section 5.2 deals with the theoretical problem to find a class of relations admitting first-order definable acceleration (modulo a first-order theory of practical interest) and having both a relevance from a practical point of view, meaning that it allows the encoding of loops of programs with arrays. We also show how to exploit acceleration in practice: the solution we propose in section 5.3 get rid of nested quantifiers, required to encode the acceleration of relations representing program loops, by including abstraction. Finally we experimentally showed that acceleration can be considered a complement of abstraction-based solutions.

Our acceleration procedure is template-based. The identification of new acceleration templates would allow a broader application of acceleration in program analysis. It would be also interesting to study a framework leveraging the results of [Bozga et al., 2009a, Bozga et al., 2009b, Bozga et al., 2010] in a modular way for finding array accelerations.

The investigation that lead us to the findings given in chapter 6 has been triggered by the interest in understanding whether acceleration would have been enough to decide the safety of programs with arrays. The paper [Bozga et al., 2014] targets the same problem in the context of programs with integers. In presence of arrays, the situation is more problematic. Constraining the shape of the control-flow graph of the program to a flat structure (i.e., every location belongs to at most one loop) and requiring that each loop of the program belongs to the fragment admitting a definable acceleration is not enough to infer decidability results, since the proof obligations (dis)proving the safety of the program might be outside decidable fragments of our background theory of

arrays ([Bradley et al., 2006,Habermehl et al., 2008b,Ge and de Moura, 2009]). Section 6.2 identifies a new decidable fragment of the mono-sorted theory of arrays while section 6.3 deals with the multi-sorted case. These fragments allow us to identify a class of programs with arrays admitting decidable reachability analysis. This is the contribution of chapter 7.

It is not excluded that the class of programs with arrays for which the safety is decidable is actually bigger than the one we found. In order to enlarge this class one may want to investigate both new acceleration schemata and new decidable quantified fragments of the theory of arrays.

A last contribution of the thesis is the tool BOOSTER. Leveraging the declarative nature of all our contributions, we studied how to integrate all of them in a single framework. In BOOSTER, abstract interpretation, acceleration and lazy abstraction with interpolation live together and collaborate to the generation of safe inductive invariants. BOOSTER does not only integrate static analysis techniques proposed in this thesis, but complement this set with state-of-the-art solutions like BMC and abstract interpretation. The abstract interpreter implemented in BOOSTER, in particular, works with the polyhedra abstract domain. It generates, therefore, *quantifier-free* invariants over $\mathcal{LIA}$. This has been done intentionally: experiments show that such quantifier-free inductive (but not safe!) invariant can be generalized by the refinement procedure and may lead to the generation of better interpolants, achieving the ultimate goal of increasing the success of BOOSTER on a larger class of examples. In the future it would be interesting to evaluate the benefits of abstract domain targeting the inference of *quantified* inductive invariants, e.g., [Cousot et al., 2011, Gulwani et al., 2008]. This is not straightforward: a quantified inductive invariant would add universal quantifiers in the guard of the transitions. This means that the LAWI framework should include some techniques to deal with these extra quantifiers, as it has been done in section 5.3 to deal with accelerated transitions.

Despite the tool BOOSTER, implementing and integrating all the contribution of this thesis, showed good practical results, many challenges, some of which have been highlighted in this chapter, are still open. Their solutions will likely improve the state-of-the-art of verification of program with arrays and may positively influence the wider area of parameterized systems verification.

Bibliography

[Abdulla, 2010] Abdulla, P. (2010). Forcing monotonicity in parameterized verification: From multisets to words. In van Leeuwen, J., Muscholl, A., Peleg, D., Pokorný, J., and Rumpe, B., editors, *SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 23-29, 2010. Proceedings*, volume 5901 of *Lecture Notes in Computer Science*, pages 1–15. Springer.

[Abdulla et al., 2009] Abdulla, P., Atto, M., Cederberg, J., and Ji, R. (2009). Automated analysis of data-dependent programs with dynamic memory. In Liu, Z. and Ravn, A., editors, *Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings*, volume 5799 of *Lecture Notes in Computer Science*, pages 197–212. Springer.

[Abdulla et al., 2008a] Abdulla, P., Bouajjani, A., Cederberg, J., Haziza, F., and Rezine, A. (2008a). Monotonic abstraction for programs with dynamic memory heaps. In Gupta, A. and Malik, S., editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 341–354. Springer.

[Abdulla et al., 1996] Abdulla, P., Cerans, K., Jonsson, B., and Tsay, Y.-K. (1996). General decidability theorems for infinite-state systems. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*, pages 313–321. IEEE Computer Society.

[Abdulla et al., 2007a] Abdulla, P., Delzanno, G., Henda, N. B., and Rezine, A. (2007a). Regular model checking without transducers (on efficient ver-

ification of parameterized systems). In [Grumberg and Huth, 2007], pages 721–736.

[Abdulla et al., 2007b] Abdulla, P., Delzanno, G., and Rezine, A. (2007b). Parameterized verification of infinite-state processes with global conditions. In [Damm and Hermanns, 2007], pages 145–157.

[Abdulla et al., 2008b] Abdulla, P., Henda, N. B., Delzanno, G., and Rezine, A. (2008b). Handling parameterized systems with non-atomic global conditions. In [Logozzo et al., 2008], pages 22–36.

[Abdulla and Jonsson, 1996] Abdulla, P. and Jonsson, B. (1996). Verifying programs with unreliable channels. *Inf. Comput.*, 127(2):91–101.

[Ábrahám and Havelund, 2014] Ábrahám, E. and Havelund, K., editors (2014). *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*. Springer.

[ACMs Press Release on the 2007 A.M. Turing Award recipients., 2007] ACMs Press Release on the 2007 A.M. Turing Award recipients. (2007). `http://www.acm.org/press-room/news-releases-2008/turing-award-07/`.

[Aho et al., 2007] Aho, A., Lam, M., Sethi, R., and Ullman, J. (2007). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Educational Publishers, Incorporated.

[Albarghouthi et al., 2012a] Albarghouthi, A., Gurfinkel, A., and Chechik, M. (2012a). Craig interpretation. In [Miné and Schmidt, 2012], pages 300–316.

[Albarghouthi et al., 2012b] Albarghouthi, A., Li, Y., Gurfinkel, A., and Chechik, M. (2012b). UFO: A framework for abstraction- and interpolation-based software verification. In [Madhusudan and Seshia, 2012], pages 672–678.

[Alberti, 2010] Alberti, F. (2010). Verifica parametrica di protocolli fault-tolerant. Master's thesis, Università degli Studi di Milano.

[Alberti et al., 2011a] Alberti, F., Armando, A., and Ranise, S. (2011a). ASASP: automated symbolic analysis of security policies. In Bjørner, N. and Sofronie-Stokkermans, V., editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 26–33. Springer.

[Alberti et al., 2011b] Alberti, F., Armando, A., and Ranise, S. (2011b). Efficient symbolic automated analysis of administrative attribute-based rbac-policies. In Cheung, B., Hui, L. C. K., Sandhu, R., and Wong, D., editors, *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, pages 165–175. ACM.

[Alberti et al., 2012a] Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., and Sharygina, N. (2012a). Lazy abstraction with interpolants for arrays. In Bjørner, N. and Voronkov, A., editors, *LPAR*, volume 7180 of *Lecture Notes in Computer Science*, pages 46–61. Springer.

[Alberti et al., 2012b] Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., and Sharygina, N. (2012b). Reachability Modulo Theory library. In *10th International Workshop on Satisfiability Modulo Theories (SMT)*.

[Alberti et al., 2012c] Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., and Sharygina, N. (2012c). SAFARI: SMT-Based Abstraction for Arrays with Interpolants. In [Madhusudan and Seshia, 2012], pages 679–685.

[Alberti et al., 2014a] Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., and Sharygina, N. (2014a). An extension of lazy abstraction with interpolation for programs with arrays. *Formal Methods in System Design*, 45(1):63–109.

[Alberti et al., 2010a] Alberti, F., Ghilardi, S., Pagani, E., Ranise, S., and Rossi, G. (2010a). Automated support for the design and validation of fault tolerant parameterized systems: a case study. *ECEASST*, 35.

[Alberti et al., 2010b] Alberti, F., Ghilardi, S., Pagani, E., Ranise, S., and Rossi, G. (2010b). Brief announcement: Automated support for the design and validation of fault tolerant parameterized systems - a case study. In Lynch, N. and Shvartsman, A. A., editors, *DISC*, volume 6343 of *Lecture Notes in Computer Science*, pages 392–394. Springer.

[Alberti et al., 2012d] Alberti, F., Ghilardi, S., Pagani, E., Ranise, S., and Rossi, G. (2012d). Universal guards, relativization of quantifiers, and failure models in Model Checking Modulo Theories. *JSAT*, 8(1/2):29–61.

[Alberti et al., 2013a] Alberti, F., Ghilardi, S., and Sharygina, N. (2013a). Acceleration-based safety decision procedure for programs with arrays. In McMillan, K. L., Middeldorp, A., Sutcliffe, G., and Voronkov, A., editors, *LPAR 2013, 19th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, December 12-17, 2013, Stellenbosch, South Africa, Short papers proceedings*, volume 26 of *EPiC Series*, pages 1–8. EasyChair.

[Alberti et al., 2013b] Alberti, F., Ghilardi, S., and Sharygina, N. (2013b). Definability of accelerated relations in a theory of arrays and its applications. In Fontaine, P., Ringeissen, C., and Schmidt, R. A., editors, *Frontiers of Combining Systems - 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings*, volume 8152 of *Lecture Notes in Computer Science*, pages 23–39. Springer.

[Alberti et al., 2014b] Alberti, F., Ghilardi, S., and Sharygina, N. (2014b). Booster: An acceleration-based verification framework for array programs. In Cassez, F. and Raskin, J., editors, *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, volume 8837 of *Lecture Notes in Computer Science*, pages 18–23. Springer.

[Alberti et al., 2014c] Alberti, F., Ghilardi, S., and Sharygina, N. (2014c). Decision procedures for Flat Array Properties. In [Ábrahám and Havelund, 2014], pages 15–30.

[Alberti et al., 2014d] Alberti, F., Ghilardi, S., and Sharygina, N. (2014d). A framework for the verification of parameterized infinite-state systems. In Giordano, L., Gliozzi, V., and Pozzato, G. L., editors, *Proceedings of the 29th Italian Conference on Computational Logic, Torino, Italy, June 16-18, 2014.*, volume 1195 of *CEUR Workshop Proceedings*, pages 303–308. CEUR-WS.org.

[Alberti et al., 2015] Alberti, F., Ghilardi, S., and Sharygina, N. (2015). Decision procedures for Flat Array Properties. *Journal of Automated Reasoning*, 54(4):327–352.

[Alberti and Monniaux, 2015] Alberti, F. and Monniaux, D. (2015). Polyhedra to the rescue of array interpolants. In *SAC 2015*. To appear.

[Alberti and Sharygina, 2012] Alberti, F. and Sharygina, N. (2012). Invariant generation by infinite-state model checking. In *2nd International Workshop on Intermediate Verification Languages*.

[Armando et al., 2007a] Armando, A., Benerecetti, M., Carotenuto, D., Mantovani, J., and Spica, P. (2007a). The Eureka tool for software model checking. In Stirewalt, R., Egyed, A., and Fischer, B., editors, *ASE*, pages 541–542. ACM.

[Armando et al., 2007b] Armando, A., Benerecetti, M., and Mantovani, J. (2007b). Abstraction refinement of linear programs with arrays. In [Grumberg and Huth, 2007], pages 373–388.

[Bach and Shallit, 1996] Bach, E. and Shallit, J. (1996). *Algorithmic number theory. Vol. 1.* Foundations of Computing Series. MIT Press.

[Bagnara et al., 2005] Bagnara, R., Hill, P., Mazzi, E., and Zaffanella, E. (2005). Widening operators for weakly-relational numeric abstractions. In [Hankin and Siveroni, 2005], pages 3–18.

[Bagnara et al., 2003] Bagnara, R., Hill, P., Ricci, E., and Zaffanella, E. (2003). Precise widening operators for convex polyhedra. In Cousot, R., editor, *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 337–354. Springer.

[Ball and Rajamani, 2002] Ball, T. and Rajamani, S. (2002). The SLAM project: debugging system software via static analysis. In [Launchbury and Mitchell, 2002], pages 1–3.

[Behrmann et al., 2002] Behrmann, G., Bengtsson, J., David, A., Larsen, K., Pettersson, P., and Yi, W. (2002). UPPAAL implementation secrets. In Damm, W. and Olderog, E.-R., editors, *FTRTFT*, volume 2469 of *Lecture Notes in Computer Science*, pages 3–22. Springer.

[Beyer, 2013] Beyer, D. (2013). Second Competition on Software Verification - (summary of SV-COMP 2013). In Piterman, N. and Smolka, S., editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 594–609. Springer.

[Beyer, 2014] Beyer, D. (2014). Status report on software verification - (competition summary SV-COMP 2014). In [Ábrahám and Havelund, 2014], pages 373–388.

[Beyer et al., 2007a] Beyer, D., Henzinger, T. A., Jhala, R., and Majumdar, R. (2007a). The software model checker Blast. *STTT*, 9(5-6):505–525.

[Beyer et al., 2007b] Beyer, D., Henzinger, T. A., Majumdar, R., and Rybalchenko, A. (2007b). Invariant synthesis for combined theories. In Cook, B. and Podelski, A., editors, *VMCAI*, volume 4349 of *Lecture Notes in Computer Science*, pages 378–394. Springer.

[Beyer and Keremoglu, 2011] Beyer, D. and Keremoglu, E. (2011). CPAchecker: A tool for configurable software verification. In Gopalakrishnan, G. and Qadeer, S., editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer.

[Biere and Bloem, 2014] Biere, A. and Bloem, R., editors (2014). *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*. Springer.

[Biere et al., 1999] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999). Symbolic model checking without BDDs. In Cleaveland, R., editor, *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer.

[Birgmeier et al., 2014] Birgmeier, J., Bradley, A., and Weissenbacher, G. (2014). Counterexample to induction-guided abstraction-refinement (CTIGAR). In [Biere and Bloem, 2014], pages 831–848.

[Bjesse and Slobodová, 2011] Bjesse, P. and Slobodová, A., editors (2011). *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*. FMCAD Inc.

[Bjørner, 2010] Bjørner, N. (2010). Linear quantifier elimination as an abstract decision procedure. In [Giesl and Hähnle, 2010], pages 316–330.

[Bjørner et al., 2013] Bjørner, N., McMillan, K., and Rybalchenko, A. (2013). On solving universally quantified horn clauses. In [Logozzo and Fähndrich, 2013], pages 105–125.

[Blanchet et al., 2002] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2002). Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In Mogensen, T., Schmidt, D., and Sudborough, I. H., editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 85–108. Springer.

[Börger et al., 1997] Börger, E., Grädel, E., and Gurevich, Y. (1997). *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer.

[Bouajjani and Maler, 2009] Bouajjani, A. and Maler, O., editors (2009). *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*. Springer.

[Bouton et al., 2009] Bouton, T., de Oliveira, D. C. B., Déharbe, D., and Fontaine, P. (2009). veriT: An open, trustable and efficient smt-solver. In Schmidt, R., editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer.

[Bozga et al., 2009a] Bozga, M., Gîrlea, C., and Iosif, R. (2009a). Iterating octagons. In Kowalewski, S. and Philippou, A., editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 337–351. Springer.

[Bozga et al., 2009b] Bozga, M., Habermehl, P., Iosif, R., Konecný, F., and Vojnar, T. (2009b). Automatic verification of integer array programs. In [Bouajjani and Maler, 2009], pages 157–172.

[Bozga et al., 2010] Bozga, M., Iosif, R., and Konecný, F. (2010). Fast acceleration of ultimately periodic relations. In Touili, T., Cook, B., and Jackson, P., editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 227–242. Springer.

[Bozga et al., 2014] Bozga, M., Iosif, R., and Konecný, F. (2014). Safety problems are np-complete for flat integer programs with octagonal loops. In McMillan, K. and Rival, X., editors, *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*, pages 242–261. Springer.

[Bozga et al., 2009c] Bozga, M., Iosif, R., and Lakhnech, Y. (2009c). Flat parametric counter automata. *Fundam. Inform.*, 91(2):275–303.

[Bradley, 2011] Bradley, A. (2011). SAT-based model checking without unrolling. In Jhala, R. and Schmidt, D., editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer.

[Bradley et al., 2006] Bradley, A., Manna, Z., and Sipma, H. (2006). What's decidable about arrays? In Emerson, E. and Namjoshi, K., editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer.

[Brillout et al., 2010] Brillout, A., Kroening, D., Rümmer, P., and Wahl, T. (2010). An interpolating sequent calculus for quantifier-free Presburger arithmetic. In [Giesl and Hähnle, 2010], pages 384–399.

[Bruttomesso et al., 2006] Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Santuari, A., and Sebastiani, R. (2006). To ackermann-ize or not to ackermann-ize? on efficiently handling uninterpreted function symbols in $SMT(EUF \grave{e} T)$. In Hermann, M. and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, volume 4246 of *Lecture Notes in Computer Science*, pages 557–571. Springer.

[Bruttomesso et al., 2012a] Bruttomesso, R., Ghilardi, S., and Ranise, S. (2012a). From strong amalgamability to modularity of quantifier-free interpolation. In *IJCAR*, Lecture Notes in Computer Science, pages 118–133. Springer.

[Bruttomesso et al., 2012b] Bruttomesso, R., Ghilardi, S., and Ranise, S. (2012b). Quantifier-free interpolation of a theory of arrays. *Logical Methods in Computer Science*, 8(2).

[Bruttomesso et al., 2010] Bruttomesso, R., Pek, E., Sharygina, N., and Tsitovich, A. (2010). The OpenSMT solver. In Esparza, J. and Majumdar, R., editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 150–153. Springer.

[Carioni et al., 2011] Carioni, A., Ghilardi, S., and Ranise, S. (2011). Automated termination in Model Checking Modulo Theories. In Delzanno, G. and Potapov, I., editors, *RP*, volume 6945 of *Lecture Notes in Computer Science*, pages 110–124. Springer.

[Chandra and Toueg, 1990] Chandra, T. and Toueg, S. (1990). Time and message efficient reliable broadcasts. In van Leeuwen, J. and Santoro, N., editors, *Distributed Algorithms, 4th International Workshop, WDAG '90, Bari, Italy, September 24-26, 1990, Proceedings*, volume 486 of *Lecture Notes in Computer Science*, pages 289–303. Springer.

[Chase et al., 1990] Chase, D., Wegman, M., and Zadeck, F. (1990). Analysis of pointers and structures. In Fischer, B., editor, *PLDI*, pages 296–310. ACM.

[Cimatti and Griggio, 2012] Cimatti, A. and Griggio, A. (2012). Software model checking via IC3. In [Madhusudan and Seshia, 2012], pages 277–293.

[Cimatti et al., 2014] Cimatti, A., Griggio, A., Mover, S., and Tonetta, S. (2014). IC3 modulo theories via implicit predicate abstraction. In [Ábrahám and Havelund, 2014], pages 46–61.

[Cimatti et al., 2010] Cimatti, A., Griggio, A., and Sebastiani, R. (2010). Efficient generation of Craig interpolants in Satisfiability Modulo Theories. *ACM Trans. Comput. Log.*, 12(1):7.

[Clarisó and Cortadella, 2007] Clarisó, R. and Cortadella, J. (2007). The octahedron abstract domain. *Sci. Comput. Program.*, 64(1):115–139.

[Clarke et al., 2000] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000). Counterexample-guided abstraction refinement. In Emerson, E. and Sistla, A., editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer.

[Clarke et al., 2004] Clarke, E., Kroening, D., and Lerda, F. (2004). A tool for checking ANSI-C programs. In Jensen, K. and Podelski, A., editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer.

[Clarke et al., 2001] Clarke, E. M., Grumberg, O., and Peled, D. (2001). *Model checking*. MIT Press.

[Comon and Jurski, 1998] Comon, H. and Jurski, Y. (1998). Multiple counters automata, safety analysis and Presburger arithmetic. In Hu, A. and Vardi, M., editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 268–279. Springer.

[Cook, 1978] Cook, S. A. (1978). Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90.

[Cooper, 1972] Cooper, D. (1972). Theorem proving in arithmetic without multiplication. In Meltzer, B. and Michie, D., editors, *Machine Intelligence*, volume 7, pages 91–100. Edinburgh University Press.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Graham, R., Harrison, M., and Sethi, R., editors, *POPL*, pages 238–252. ACM.

[Cousot et al., 2005] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2005). The astreé analyzer. In Sagiv, S., editor, *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer.

[Cousot et al., 2011] Cousot, P., Cousot, R., and Logozzo, F. (2011). A parametric segmentation functor for fully automatic and scalable array content analysis. In Ball, T. and Sagiv, M., editors, *POPL*, pages 105–118. ACM.

[Cousot and Halbwachs, 1978] Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear restraints among variables of a program. In Aho, A., Zilles, S., and Szymanski, T., editors, *POPL*, pages 84–96. ACM Press.

[Craig, 1957] Craig, W. (1957). Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285.

[Damm and Hermanns, 2007] Damm, W. and Hermanns, H., editors (2007). *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*. Springer.

[De Angelis et al., 2014a] De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. (2014a). Program verification via iterated specialization. *Sci. Comput. Program.*, 95:149–175.

[De Angelis et al., 2014b] De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. (2014b). Verimap: A tool for verifying programs through transformations. In [Ábrahám and Havelund, 2014], pages 568–574.

[de Moura and Bjørner, 2007] de Moura, L. and Bjørner, N. (2007). Efficient e-matching for SMT solvers. In Pfenning, F., editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer.

[de Moura and Bjørner, 2008] de Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In [Ramakrishnan and Rehof, 2008], pages 337–340.

[de Moura and Bjørner, 2009] de Moura, L. and Bjørner, N. (2009). Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52. IEEE.

[de Moura and Bjørner, 2011] de Moura, L. and Bjørner, N. (2011). Satisfiability Modulo Theories: introduction and applications. *Commun. ACM*, 54(9):69–77.

[Delzanno et al., 1999] Delzanno, G., Esparza, J., and Podelski, A. (1999). Constraint-based analysis of broadcast protocols. In Flum, J. and Rodríguez-Artalejo, M., editors, *CSL*, volume 1683 of *Lecture Notes in Computer Science*, pages 50–66. Springer.

[Detlefs et al., 2003] Detlefs, D., Nelson, G., and Saxe, J. (2003). Simplify: a theorem prover for program checking. Technical Report HPL-2003-148, HP Labs.

[Dillig et al., 2010] Dillig, I., Dillig, T., and Aiken, A. (2010). Fluid updates: Beyond strong vs. weak updates. In Gordon, A., editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 246–266. Springer.

[Dimitrova and Podelski, 2008] Dimitrova, R. and Podelski, A. (2008). Is lazy abstraction a decision procedure for broadcast protocols? In [Logozzo et al., 2008], pages 98–111.

[Dragan and Kovács, 2014] Dragan, I. and Kovács, L. (2014). LINGVA: Generating and proving program properties using symbol elimination. In *PSI*. To appear.

[Dudka et al., 2011] Dudka, K., Peringer, P., and Vojnar, T. (2011). Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *CAV*, pages 372–378.

[Dudka et al., 2013] Dudka, K., Peringer, P., and Vojnar, T. (2013). Byte-precise verification of low-level list manipulation. In [Logozzo and Fähndrich, 2013], pages 215–237.

[Eén et al., 2011] Eén, N., Mishchenko, A., and Brayton, R. (2011). Efficient implementation of property directed reachability. In [Bjesse and Slobodová, 2011], pages 125–134.

[Esparza et al., 1999] Esparza, J., Finkel, A., and Mayr, R. (1999). On the verification of broadcast protocols. In *LICS*, pages 352–359. IEEE Computer Society.

[Finkel and Leroux, 2002] Finkel, A. and Leroux, J. (2002). How to compose Presburger-accelerations: Applications to broadcast protocols. In Agrawal, M. and Seth, A., editors, *FSTTCS*, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156. Springer.

[Flanagan and Qadeer, 2002] Flanagan, C. and Qadeer, S. (2002). Predicate abstraction for software verification. In [Launchbury and Mitchell, 2002], pages 191–202.

[Floyd, 1962] Floyd, R. (1962). Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345.

[Furia and Meyer, 2010] Furia, C. and Meyer, B. (2010). Inferring loop invariants using postconditions. In Blass, A., Dershowitz, N., and Reisig, W., editors, *Fields of Logic and Computation*, volume 6300 of *Lecture Notes in Computer Science*, pages 277–300. Springer.

[Garg et al., 2014] Garg, P., Löding, C., Madhusudan, P., and Neider, D. (2014). ICE: A robust framework for learning invariants. In [Biere and Bloem, 2014], pages 69–87.

[Ge et al., 2009] Ge, Y., Barrett, C. W., and Tinelli, C. (2009). Solving quantified verification conditions using satisfiability modulo theories. *Annals of Mathematics and Artificial Intelligence*, 55(1-2):101–122.

[Ge and de Moura, 2009] Ge, Y. and de Moura, L. (2009). Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In [Bouajjani and Maler, 2009], pages 306–320.

[Ghilardi and Ranise, 2009] Ghilardi, S. and Ranise, S. (2009). Model Checking Modulo Theory at work: the integration of Yices in MCMT. In *AFM*.

[Ghilardi and Ranise, 2010a] Ghilardi, S. and Ranise, S. (2010a). Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *Logical Methods in Computer Science*, 6(4).

[Ghilardi and Ranise, 2010b] Ghilardi, S. and Ranise, S. (2010b). Mcmt: A model checker modulo theories. In [Giesl and Hähnle, 2010], pages 22–29.

[Ghilardi et al., 2009] Ghilardi, S., Ranise, S., and Valsecchi, T. (2009). Light-weight SMT-based model checking. *Electr. Notes Theor. Comput. Sci.*, 250(2):85–102.

[Giesl and Hähnle, 2010] Giesl, J. and Hähnle, R., editors (2010). *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*. Springer.

[Goel et al., 2008] Goel, A., Krstic, S., and Fuchs, A. (2008). Deciding Array formulas with fruagal axiom instantiation. In *SMT Workshop 2008*.

[Gopan et al., 2005] Gopan, D., Reps, T. W., and Sagiv, S. (2005). A framework for numeric analysis of array operations. In Palsberg, J. and Abadi, M., editors, *POPL*, pages 338–350. ACM.

[Graf and Saïdi, 1997] Graf, S. and Saïdi, H. (1997). Construction of abstract state graphs with PVS. In Grumberg, O., editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer.

[Grumberg and Huth, 2007] Grumberg, O. and Huth, M., editors (2007). *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*. Springer.

[Gulwani et al., 2008] Gulwani, S., McCloskey, B., and Tiwari, A. (2008). Lifting abstract interpreters to quantified logical domains. In Necula, G. and Wadler, P., editors, *POPL*, pages 235–246. ACM.

[Gulwani and Tiwari, 2006] Gulwani, S. and Tiwari, A. (2006). Combining abstract interpreters. In Schwartzbach, M. and Ball, T., editors, *PLDI*, pages 376–386. ACM.

[Gurfinkel et al., 2011] Gurfinkel, A., Chaki, S., and Sapra, S. (2011). Efficient predicate abstraction of program summaries. In Bobaru, M. G., Havelund, K., Holzmann, G., and Joshi, R., editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20,*

2011. Proceedings, volume 6617 of *Lecture Notes in Computer Science*, pages 131–145. Springer.

[Habermehl et al., 2008a] Habermehl, P., Iosif, R., and Vojnar, T. (2008a). A logic of singly indexed arrays. In Cervesato, I., Veith, H., and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, volume 5330 of *Lecture Notes in Computer Science*, pages 558–573. Springer.

[Habermehl et al., 2008b] Habermehl, P., Iosif, R., and Vojnar, T. (2008b). What else is decidable about integer arrays? In Amadio, R., editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, volume 4962 of *Lecture Notes in Computer Science*, pages 474–489. Springer.

[Halbwachs and Péron, 2008] Halbwachs, N. and Péron, M. (2008). Discovering properties about arrays in simple programs. In Gupta, R. and Amarasinghe, S., editors, *PLDI*, pages 339–348. ACM.

[Hankin and Siveroni, 2005] Hankin, C. and Siveroni, I., editors (2005). *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*. Springer.

[Hendriks and Larsen, 2002] Hendriks, M. and Larsen, K. (2002). Exact acceleration of real-time model checking. *Electr. Notes Theor. Comput. Sci.*, 65(6):120–139.

[Henry et al., 2012] Henry, J., Monniaux, D., and Moy, M. (2012). Succinct representations for abstract interpretation - combined analysis algorithms and experimental evaluation. In [Miné and Schmidt, 2012], pages 283–299.

[Henzinger et al., 2004] Henzinger, T., Jhala, R., Majumdar, R., and McMillan, K. (2004). Abstractions from proofs. In Jones, N. and Leroy, X., editors, *POPL*, pages 232–244. ACM.

[Henzinger et al., 2002] Henzinger, T., Jhala, R., Majumdar, R., and Sutre, G. (2002). Lazy abstraction. In [Launchbury and Mitchell, 2002], pages 58–70.

[Hind, 2001] Hind, M. (2001). Pointer analysis: haven't we solved this problem yet? In Field, J. and Snelting, G., editors, *PASTE*, pages 54–61. ACM.

[Hoder and Bjørner, 2012] Hoder, K. and Bjørner, N. (2012). Generalized Property Directed Reachability. In Cimatti, A. and Sebastiani, R., editors, *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer.

[Hoder et al., 2010] Hoder, K., Kovács, L., and Voronkov, A. (2010). Interpolation and symbol elimination in Vampire. In [Giesl and Hähnle, 2010], pages 188–195.

[Hoder et al., 2011] Hoder, K., Kovács, L., and Voronkov, A. (2011). Invariant generation in Vampire. In Abdulla, P. and Leino, K., editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 60–64. Springer.

[Hodges, 1993] Hodges, W. (1993). *Model Theory*, volume 42 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, Cambridge.

[Hojjat et al., 2012] Hojjat, H., Iosif, R., Konecný, F., Kuncak, V., and Rümmer, P. (2012). Accelerating interpolants. In Chakraborty, S. and Mukund, M., editors, *ATVA*, volume 7561 of *Lecture Notes in Computer Science*, pages 187–202. Springer.

[Ihlemann et al., 2008] Ihlemann, C., Jacobs, S., and Sofronie-Stokkermans, V. (2008). On local reasoning in verification. In [Ramakrishnan and Rehof, 2008], pages 265–281.

[Jhala and McMillan, 2006] Jhala, R. and McMillan, K. (2006). A practical and complete approach to predicate refinement. In Hermanns, H. and Palsberg, J., editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer.

[Jhala and McMillan, 2007] Jhala, R. and McMillan, K. (2007). Array abstractions from proofs. In [Damm and Hermanns, 2007], pages 193–206.

[Kapur et al., 2006] Kapur, D., Majumdar, R., and Zarba, C. (2006). Interpolation for data structures. In Young, M. and Devanbu, P., editors, *SIGSOFT FSE*, pages 105–116. ACM.

[Kovács and Voronkov, 2009] Kovács, L. and Voronkov, A. (2009). Finding loop invariants for programs over arrays using a theorem prover. In Chechik,

M. and Wirsing, M., editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 470–485. Springer.

[Lahiri and Bryant, 2004a] Lahiri, S. and Bryant, R. (2004a). Constructing quantified invariants via predicate abstraction. In Steffen, B. and Levi, G., editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 267–281. Springer.

[Lahiri and Bryant, 2004b] Lahiri, S. and Bryant, R. (2004b). Indexed predicate discovery for unbounded system verification. In Alur, R. and Peled, D., editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 135–147. Springer.

[Larraz et al., 2013] Larraz, D., Rodríguez-Carbonell, E., and Rubio, A. (2013). SMT-based array invariant generation. In Giacobazzi, R., Berdine, J., and Mastroeni, I., editors, *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 169–188. Springer.

[Launchbury and Mitchell, 2002] Launchbury, J. and Mitchell, J., editors (2002). *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*. ACM.

[Lecomte, 2008] Lecomte, T. (2008). Safe and reliable metro platform screen doors control/command systems. In Cuéllar, J., Maibaum, T., and Sere, K., editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 430–434. Springer.

[Lewis, 1978] Lewis, H. (1978). Complexity of solvable cases of the decision problem for the predicate calculus. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 35–47. IEEE Computer Society.

[Logozzo and Fähndrich, 2010] Logozzo, F. and Fähndrich, M. (2010). Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Science of Computer Programming*, 75(9):796–807.

[Logozzo and Fähndrich, 2013] Logozzo, F. and Fähndrich, M., editors (2013). *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*. Springer.

[Logozzo et al., 2008] Logozzo, F., Peled, D., and Zuck, L., editors (2008). *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings*, volume 4905 of *Lecture Notes in Computer Science*. Springer.

[Lynch, 1996] Lynch, N. (1996). *Distributed Algorithms*. Morgan Kaufmann.

[Madhusudan and Seshia, 2012] Madhusudan, P. and Seshia, S., editors (2012). *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*. Springer.

[Manna and Pnueli, 1995] Manna, Z. and Pnueli, A. (1995). *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag.

[McCarthy, 1962] McCarthy, J. (1962). Towards a mathematical science of computation. In *International Federation for Information Processing Congress*, pages 21–28.

[McMillan, 2006] McMillan, K. (2006). Lazy abstraction with interpolants. In Ball, T. and Jones, R., editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer.

[McMillan, 2008] McMillan, K. (2008). Quantified invariant generation using an interpolating saturation prover. In [Ramakrishnan and Rehof, 2008], pages 413–427.

[McMillan, 2011] McMillan, K. (2011). Interpolants from Z3 proofs. In [Bjesse and Slobodová, 2011], pages 19–27.

[Mendelson, 1997] Mendelson, E. (1997). *Introduction to Mathematical Logic*. Taylor & Francis.

[Miné, 2006] Miné, A. (2006). The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100.

[Miné and Schmidt, 2012] Miné, A. and Schmidt, D., editors (2012). *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, volume 7460 of *Lecture Notes in Computer Science*. Springer.

[Minsky, 1967] Minsky, M. (1967). *Computation: finite and infinite machines*. Prentice-Hall series in automatic computation. Prentice-Hall.

[Nelson and Oppen, 1979] Nelson, G. and Oppen, D. (1979). Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57.

[Newcombe, 2014] Newcombe, C. (2014). Why amazon chose TLA +. In Ameur, Y. and Schewe, K., editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science*, pages 25–39. Springer.

[Nieuwenhuis and Oliveras, 2005] Nieuwenhuis, R. and Oliveras, A. (2005). DPLL(T) with exhaustive theory propagation and its application to difference logic. In Etessami, K. and Rajamani, S., editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 321–334. Springer.

[Oppen, 1978] Oppen, D. (1978). A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *J. Comput. Syst. Sci.*, 16(3):323–332.

[Papadimitriou, 1981] Papadimitriou, C. (1981). On the complexity of integer programming. *J. ACM*, 28(4):765–768.

[Podelski and Wies, 2005] Podelski, A. and Wies, T. (2005). Boolean heaps. In [Hankin and Siveroni, 2005], pages 268–283.

[Ramakrishnan and Rehof, 2008] Ramakrishnan, C. and Rehof, J., editors (2008). *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*. Springer.

[Ranise and Tinelli, 2006] Ranise, S. and Tinelli, C. (2006). The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.smt-lib.org`.

[Reynolds et al., 2013] Reynolds, A., Tinelli, C., Goel, A., Krstic, S., Deters, M., and Barrett, C. (2013). Quantifier instantiation techniques for finite model finding in SMT. In Bonacina, M., editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 377–391. Springer.

[Reynolds, 2002] Reynolds, J. (2002). Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society.

[Rollini et al., 2013] Rollini, S., Alt, L., Fedyukovich, G., Hyvärinen, A., and Sharygina, N. (2013). Periplo: A framework for producing effective interpolants in sat-based software verification. In McMillan, K., Middeldorp, A., and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *Lecture Notes in Computer Science*, pages 683–693. Springer.

[Rosser, 1939] Rosser, J. (1939). The n-th prime is greater than $n \log n$. *Proceedings of the London Mathematical Society*, s2-45(1):21–44.

[RTCA, 2011] RTCA (2011). DO-333, Formal Methods Supplement to DO-178C and DO-278A.

[Rümmer and Subotic, 2013] Rümmer, P. and Subotic, P. (2013). Exploring interpolants. In Jobstmann, B. and Ray, S., editors, *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 69–76. IEEE.

[Sagiv et al., 1999] Sagiv, S., Reps, T., and Wilhelm, R. (1999). Parametric shape analysis via 3-valued logic. In Appel, A. and Aiken, A., editors, *POPL*, pages 105–118. ACM.

[Sebastiani, 2007] Sebastiani, R. (2007). Lazy satisability modulo theories. *JSAT*, 3(3-4):141–224.

[Seghir et al., 2009] Seghir, M., Podelski, A., and Wies, T. (2009). Abstraction refinement for quantified array assertions. In Palsberg, J. and Su, Z., editors, *SAS*, volume 5673 of *Lecture Notes in Computer Science*, pages 3–18. Springer.

[Semënov, 1984] Semënov, A. (1984). Logical theories of one-place functions on the set of natural numbers. *Izvestiya: Mathematics*, 22:587–618.

[Srivastava and Gulwani, 2009] Srivastava, S. and Gulwani, S. (2009). Program verification using templates over predicate abstraction. In Hind, M. and Diwan, A., editors, *PLDI*, pages 223–234. ACM.

[Totla and Wies, 2013] Totla, N. and Wies, T. (2013). Complete instantiation-based interpolation. In Giacobazzi, R. and Cousot, R., editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 537–548. ACM.

[Turing, 1936] Turing, A. (1936). On computable numbers, with an application to the eintscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1936-7):230–265.

[Wirth, 1978] Wirth, N. (1978). *Algorithms + Data Structures = Programs*. Prentice-Hall Series in Automatic Computation. Pearson Education.