J Autom Reasoning (2015) 54:327–352
DOI 10.1007/s10817-015-9323-7

Decision Procedures for Flat Array Properties

Francesco Alberti · Silvio Ghilardi · Natasha Sharygina

Received: 19 May 2014 / Accepted: 27 February 2015 / Published online: 20 March 2015

Abstract We present new decidability results for quantified fragments of theories of arrays. Our decision procedures are parametric in the theories of indexes and elements and orthogonal with respect to known results. We show that transitive closures ('acceleration') of relation expressing certain array updates produce formulas inside our fragment; this observation will be used to identify a class of programs handling arrays having decidable reachability problem.

Keywords Decision procedures · Quantifiers · Arrays · SMT

1 Introduction

The demand for efficient decision procedures has enormously increased in the last decades, as witnessed by the success of the SMT-LIB initiative and the excellent performances of current SMT-solvers in dealing with various verification tasks related not only to hardware but also to software systems of different nature (sequential, distributed, hybrid, etc.). Decision procedures constitute, nowadays, one of the fundamental components of tools and algorithms developed for formal methods applications.

Given the complex nature of concrete problems, it is important to develop decision procedures not only for numerical domains, but also for theories modeling container data

This paper extends material previously published in [6].

F. Alberti (✉) · N. Sharygina
Faculty of Informatics, Università della Svizzera Italiana, via G. Buffi, 13, CH-6904, Lugano, Switzerland
e-mail: francesco.alberti@usi.ch

N. Sharygina
e-mail: natasha.sharygina@usi.ch

S. Ghilardi
Department of Mathematics, Università degli Studi di Milano, via C. Saldini, 50, I-20133, Milano, Italy
e-mail: silvio.ghilardi@unimi.it

structures, like arrays, lists, stacks, records, etc. For these theories it is usually impossible to establish a useful quantifier-elimination result, hence genuinely quantified assertions must be handled directly. And indeed quantified formulas arise from several static analysis and verification tasks, like modeling properties of the heap, asserting frame axioms, checking user-defined assertions in the code and reasoning about parameterized systems.

In this paper we are interested in studying the decidability of quantified fragments of theories of arrays. Quantification is required over the indexes of the arrays in order to express significant properties like "the even positions of the array a have been initialized to 0" or "the values of array a are all smaller then the values of the array b", for example.

From a logical point of view, we view arrays as *free function symbols*. Strictly speaking, McCarthy theory of arrays [33] considers also built-in update operations and has explicit extensionality support; although these features seem to get lost when modeling arrays as plain free function symbols, the loss is not very relevant in the present context because we can recover them through a mild use of single quantifiers - e.g. instead of writing $a = upd(b, i, e)$ we can write the $\forall$-formula $a(i) = e \wedge \forall j \ (j \neq i \rightarrow a(j) = b(j))$. The real problem is that adding free function symbols to a theory T (with the goal of modeling array variables) may yield to undecidable extensions of widely used theories like Presburger arithmetic [29]. It is, therefore, mandatory to identify fragments of the quantified theory 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 paper is new: its key feature is the *flatness* condition. We briefly report how this condition was suggested to us by our practice in analyzing imperative programs through our model checkers MCMT [26] and SAFARI [2]. One of the main distinguishing features of our tools is the ability of synthesizing invariants involving quantifiers. In fact, whenever arrays are involved, the synthesis of good quantified invariants is necessary to solve safety problems, even in case system specification and verification tasks are modeled via ground formulas (see e.g. [31] for examples in this sense). To generate quantified invariants, we have refined the interpolation-based abstraction/refinements loop underlying our tools (an extension of the approach presented in [34]) by a special heuristics [1, 3], aimed at searching 'good' interpolants. For the heuristics to be productive, formulas must be subjected to 'flatness' limitations on dereferencing: only positions named by variables are allowed in dereferencing. In short, let $\phi(a[t], \ldots)$ be a formula involving the term $a[t]$. Instead of handling $\phi(a[t], \ldots)$ directly, we maintain the equivalent formula $\exists x \ (x = t \wedge \phi(a[x], \ldots))$. This gives the possibility to abstract out the term t while simultaneously synthesizing a genuinely quantified assertion. Thus, flatness limitations constitute a key entry for some heuristics to work.

In this paper, we show that flatness limitations can also be used to gain decidability. In fact, whereas it is trivially true that every formula can be flattened via logical equivalences introducing extra quantifiers (by the above outlined method), it is also well-known that decidability results are sensible to the shape of quantifiers prefixes in prenex normal forms. Hence, flatness limitations combined with prefix limitations may introduce meaningful restrictions and our contribution will show 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 [27] and in fact we show that this kind of limitation can be usefully combined with flatness restrictions too. We call the fragments we obtain *Flat Array Properties*; such fragments are orthogonal to the fragments already proven decidable in the literature [16, 27, 28] (we shall defer the technical comparison with these contributions to Section 7). Here we explain the *modularity* character of our

results and their *applications* to concrete decision problems for array programs annotated with assertions or postconditions.

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 T modeling both indexes and elements of the arrays. In the other one, we take into account Flat Array Properties over a theory of arrays built by connecting two theories T_I and T_E describing the structure of indexes and elements. Our decidability results are fully declarative and parametric in the theories T, T_I, T_E. For both settings, we provide sufficient conditions on T and T_I, 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 T-formulas in one case and T_I- and T_E-formulas in the other case.

We further show, as an application of our decidability results, that the safety of an interesting class of programs handling arrays or strings of unknown length is decidable. We call this class of programs $\mathsf{simple}^0_{\mathcal{A}}$ *progams*: this class covers non-recursive programs implementing for instance searching, copying, comparing, initializing, replacing and testing functions. The method we use for showing these safety results is similar to a classical method adopted in the model-checking literature for programs manipulating integer variables (see for instance [15, 17, 22]): we first assume flatness conditions on the control flow graph of the program and then we assume that transitions labeling cycles are "accelerateable". However, since we are dealing with array manipulating programs, acceleration requires specific results that we borrow from [4]. The key point is that the shape of most accelerated transitions from [4] matches the definition of our Flat Array Properties (in fact, Flat Array Properties were designed also in order to encompass such accelerated transitions for arrays).

From the experimental point of view, the situation needs to be better investigated. We leave the related task to future (more implementation-oriented) work, we shall just make some preliminary observations in the final sections of the paper. In Section 6 we tested the effectiveness of state of the art SMT-solvers in checking the satisfiability of some Flat Array Properties arising from the verification of $\mathsf{simple}^0_{\mathcal{A}}$-progams. Results show that such tools fail or timeout on some Flat Array Properties. We tried to identify some reasons for such failures and we indicate how our procedure can cope with them; at the same time, we isolate the main sources of complexity in our procedures and discuss the impact they might have on practica l benchmarks.

Plan of the paper The paper starts by recalling in Section 2 required background notions. Section 3 introduces a decision procedure for Flat Array Properties in the case of a mono-sorted theory $\mathtt{ARR}^1(T)$ generated by adding free function symbols to a theory T. Section 4 discusses a decision procedure for Flat Array Properties in the case of the multi-sorted array theory $\mathtt{ARR}^2(T_I, T_E)$ built over two theories T_I and T_E for the indexes and elements (we supply also full lower and upper complexity bounds for the case in which T_I and T_E are both Presburger arithmetic). In Section 5 we recall and adapt required notions from [4], define the class of flat^0-programs and establish the requirements for achieving the decidability of reachability analysis on some flat^0. Such requirements are instantiated in Section 5.1 in the case of $\mathsf{simple}^0_{\mathcal{A}}$-progams, array programs with flat control-flow graph admitting definable accelerations for every loop. In Section 6 we position the fragment of Flat Array Properties with respect to the actual practical capabilities of state-of-the-art SMT-solvers. Section 7

compares our results with the state of the art, in particular with the approaches of [16, 27]. In Section 8, we conclude and discuss future work covering potential implementations.

2 Background

We use lower-case latin letters $x, i, c, d, e, \ldots$ for variables; for tuples of variables we use bold face letters like **x, i, c, d, e** ... The n-th component of a tuple $\mathbf{c}$ is indicated with c_n and $|-|$ may indicate tuples length (so that we have $\mathbf{c} = c_1, \ldots, c_{|\mathbf{c}|}$). Free constants coming from skolemizations of existential quantifiers might be given the same name of the variable they skolemize. For terms, we use letters $t, u, \ldots$, with the same conventions as above; **t, u** are used for tuples of terms (however, tuples of variables are assumed to be distinct, whereas the same is not assumed for tuples of terms - this is useful for substitutions notation, see below). 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 $E(\mathbf{x})$ we denote that the syntactic expression (term, formula, tuple of terms or of formulas) E contains at most the free variables *taken from* the tuple $\mathbf{x}$. We use lower-case Greek letters $\phi, \varphi, \psi, \ldots$ for **quantifier-free** formulas and $\alpha, \beta, \ldots$ for arbitrary formulas. The notation $\phi(\mathbf{t})$ identifies a quantifier-free formula ϕ obtained from $\phi(\mathbf{x})$ by substituting the tuple of variables $\mathbf{x}$ with the tuple of terms $\mathbf{t}$.

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. Let R be a regular expression over the alphabet $\{\exists, \forall\}$. The R-class of formulas comprises all and only those prenex formulas whose prefix generates a string $Q_1 \cdots Q_n$ matched by R.

According to the SMT-LIB standard [9], a theory 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 T. Given a Σ-structure $\mathcal{M}$, we denote by $S^{\mathcal{M}}$, $f^{\mathcal{M}}$, $P^{\mathcal{M}}$, ... the interpretation in $\mathcal{M}$ of the sort S, the function symbol f, the predicate symbol P, etc. A Σ-formula α is T-satisfiable if there exists a Σ-structure $\mathcal{M}$ in $\mathcal{C}$ such that α is true in $\mathcal{M}$ under a suitable assignment to the free variables of α (in symbols, $\mathcal{M} \models \alpha$); it is T-valid (in symbols, $T \models \alpha$) if its negation is T-unsatisfiable. Two formulas α_1 and α_2 are T-equivalent if $\alpha_1 \leftrightarrow \alpha_2$ is T-valid; α_1 T-entails α_2 (in symbols, $\alpha_1 \models_T \alpha_2$) iff $\alpha_1 \rightarrow \alpha_2$ is T-valid. The satisfiability modulo the theory T ($SMT(T)$) problem amounts to establishing the T-satisfiability of quantifier-free Σ-formulas. **All theories T we consider in this paper have decidable $SMT(T)$-problem** (we recall that this property is preserved when adding free function symbols, see [23, 41]).

A theory $T = (\Sigma, \mathcal{C})$ admits *quantifier elimination* iff for any arbitrary Σ-formula $\alpha(\mathbf{x})$ it is always possible to compute a quantifier-free formula $\varphi(\mathbf{x})$ such that $T \models \forall \mathbf{x}.(\alpha(\mathbf{x}) \leftrightarrow \varphi(\mathbf{x}))$. Thus, in view of the above assumption on decidability of $SMT(T)$-problem, a theory having quantifier elimination is decidable (i.e. T-satisfiability of *every* formula is decidable). Our favorite example of a theory with quantifier elimination is *Presburger Arithmetic*, hereafter denoted with $\mathbb{P}$; this is the theory in the signature $\{0, 1, +, -, =, <\}$ augmented with infinitely many unary predicates D_k (for each integer k greater than 1). Semantically, the intended class of models for $\mathbb{P}$ contains just the structure whose support is the set of the natural numbers, where $\{0, 1, +, -, =, <\}$ have the natural interpretation and D_k is interpreted as the set of natural numbers divisible by k (these extra predicates are needed to get quantifier elimination [36]).

Although $\mathbb{P}$ represents the fragment of arithmetic mostly used in formal approaches for the static analysis of systems, we underline that there are many other fragments that have

quantifier elimination and can be quite useful; these fragments can be both weaker (like Integer Difference Logic [35]) and stronger (like the exponentiation extension of Semënov theorem [39]) than $\mathbb{P}$. Thus, the *modular* approach proposed in this Section to model arrays is not motivated just by generalization purposes, but can have practical impact.

There exist two ways of introducing arrays in a declarative setting, the mono-sorted and the multi-sorted ways. The former is more expressive because (roughly speaking) it allows to consider indexes also as elements[1], but might be computationally more difficult to handle. We discuss decidability results for both cases, starting from the mono-sorted case.

3 The Mono-sorted Case

Let $T = (\Sigma, \mathcal{C})$ be a theory; the theory $\mathtt{ARR}^1(T)$ of *arrays over* T is obtained from T by adding to it infinitely many (fresh) free unary function symbols. This means that the signature of $\mathtt{ARR}^1(T)$ is obtained from Σ by adding to it unary function symbols (we use the letters $a, a_1, a_2, \ldots$ for them) and that a structure $\mathcal{M}$ is a model of $\mathtt{ARR}^1(T)$ iff (once the interpretations of the extra function symbols are disregarded) it is a structure belonging to the original class $\mathcal{C}$.

For array theories it is useful to introduce the following notation. 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 $a_1(t_1), \ldots, a_1(t_{|\mathbf{t}|}), \ldots, a_{|\mathbf{a}|}(t_1), \ldots, a_{|\mathbf{a}|}(t_{|\mathbf{t}|})$.

$\mathtt{ARR}^1(T)$ may be highly undecidable, even when T itself is decidable (see [29]), thus it is mandatory to limit the shape of the formulas we want to try to decide. A prenex formula or a term in the signature of $\mathtt{ARR}^1(T)$ are said to be *flat* iff for every term of the kind $a(t)$ occurring in them (here a is any array constant), the sub-term t is always a variable. Notice that every formula is logically equivalent to a flat one; however the flattening transformations usually employed in the literature are based on rewriting as

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

and consequently they may alter the quantifiers prefix of a formula. Thus it must be kept in mind (when understanding the results below), that flattening transformation cannot be operated on any occurrence of a term without exiting from the class that is claimed to be decidable. When we indicate a flat quantifier-free formula with the notation $\psi(\mathbf{x}, \mathbf{a(x)})$, we mean that such a formula is obtained from a Σ-formula of the kind $\psi(\mathbf{x}, \mathbf{z})$ (i.e. from a quantifier-free Σ-formula where at most the free variables $\mathbf{x}, \mathbf{z}$ can occur) by replacing the variables $\mathbf{z}$ by the terms $\mathbf{a(x)}$.

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

[1] This is useful in the analysis of programs, when pointers to the memory (modeled as an array) are stored into array variables.

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)

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

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

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

$\mathsf{SAT}_{\mathsf{MONO}}$ transforms an $\mathtt{ARR}^1(T)$-formula F into an equisatisfiable T-formula F_3 belonging to the $\exists^*\forall\exists^*$ fragment. More precisely, it holds that F, F_1 and F_2 are equivalent formulas, 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. Flat formulas with two universally quantified variables are not decidable for satisfiability (for $T = \mathbb{P}$), as one can realize by slightly modifying for instance the last counterexample in the Appendix of [25].

Since Presburger Arithmetic is decidable (via quantifier elimination), we get in particular that

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

4 The Multi-sorted Case

We are now considering a theory of arrays parametric in the theories specifying constraints over indexes and elements of the arrays. Formally, we need two ingredient theories, $T_I = (\Sigma_I, \mathcal{C}_I)$ and $T_E = (\Sigma_E, \mathcal{C}_E)$. 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 (but extending our results to many-sorted T_E is quite straightforward): let us call INDEX the unique sort of T_I and ELEM the unique sort of T_E.

The theory $\mathtt{ARR}^2(T_I, T_E)$ *of arrays over T_I and 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 $\mathtt{ARR}^2(T_I, T_E)$ are the structures whose reducts to the symbols of sorts INDEX and ELEM are models of T_I and T_E, respectively.

Consider now an atomic formula $P(t_1, \ldots, t_n)$ in the language of $\mathtt{ARR}^2(T_I, T_E)$ (in the typical situation, P is the equality predicate). Since the predicate symbols of $\mathtt{ARR}^2(T_I, T_E)$ 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 latter case, all terms t_i have sort ELEM and 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). We say that $P(t_1, \ldots, t_n)$ is an $\mathtt{INDEX} - atom$ in the former case and that it is an $\mathtt{ELEM} - atom$ in the latter case.

When dealing with $\mathtt{ARR}^2(T_I, T_E)$, *we shall limit ourselves to quantified variables of sort* INDEX: this limitation is justified by the benchmarks arising in applications (see Section 5). If we need topmost existentially quantified variables of sort ELEM, we can model them by skolemization, i.e. by enriching T_E with free constants. A sentence in the language of $\mathtt{ARR}^2(T_I, T_E)$ is said to be *monic* iff it is in prenex form and every INDEX atom occurring in it contains at most one variable falling within the scope of a *universal* quantifier.

Example 1 Consider the following sentences:

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

The flat formula (I) is not well-typed, hence it is not allowed in $\mathtt{ARR}^2(\mathbb{P}, \mathbb{P})$; however, it is allowed in $\mathtt{ARR}^1(\mathbb{P})$. 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 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

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

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

Theorem 2 below reduces $\mathtt{ARR}^2(T_I, T_E)$-satisfiability of $\exists^*\forall^*$-monic-flat sentences to 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 formulas to be tested for satisfiability are monic

is essential[2] and we make use of this hypothesis 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 renamings, 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 advances which types M are realized (i.e. satisfied), it introduces for each of them a witnesss 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 universal quantifiers of the original formula are 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 formulas, 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., etc.

Theorem 2 *If T_I-satisfiability of $\exists^*\forall$-sentences is decidable, then $\mathtt{ARR}^2(T_I, T_E)$-satisfiability of $\exists^*\forall^*$-monic-flat sentences is decidable.*[3]

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 4.1. Correctness and completeness of $\mathsf{SAT}_{\mathsf{MULTI}}$ are split into two lemmas (Lemmas 2 and 1) to be proved in Section 4.2 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. □

4.1 The Decision Procedure $\mathsf{SAT}_{\mathsf{MULTI}}$.

The algorithm is non-deterministic: the input formula is satisfiable iff we can guess suitable data $\mathcal{T}, \mathcal{B}$ so that the formulas F_I, F_E below are satisfiable.

[2]Undecidability arises otherwise, see again for instance the Appendix of [25] for a reduction to reachability problems of Minsky machines.

[3]The reader might have noticed that (by considering the special case of formulas in which ELEM atoms do not occur), Theorem 2 has the following corollary concerning only T_I: "if T_I-satisfiability of the $\exists^*\forall$-sentences is decidable, then T_I-satisfiability of $\exists^*\forall^*$-monic-flat sentences is decidable". There is nothing wrong in this, because by help of (computationally expensive indeed!) Boolean manipulations one can check directly that $\exists^*\forall^*$-monic-flat T_I-sentences are equivalent to disjunctions of $\exists^*\forall$ T_I-sentences. In other words, the notion of being monic becomes interesting only in presence of ELEM atoms.

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 ψ is a $T_I \cup T_E$-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)

$$K = \{A(x, \mathbf{c}) \mid A(i_k, \mathbf{c}) \text{ is an } \texttt{INDEX} \text{ atom of } F\}_{1 \le k \le n} \cup \{x = c_l\}_{1 \le l \le t}$$

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 $\neg A(x, \mathbf{c}) \in M$ (thus, types are maximal Boolean-consistent sets and as such are pairwise incompatible). Guess a set $\mathcal{T} = \{M_1, \ldots, M_q\}$ 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 $\mathbf{e} = \langle e_{l,m} \rangle (1 \le l \le s, 1 \le m \le t + q)$ 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 $\mathcal{B}$ 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 $\mathcal{B}$.

STEP V. Check the 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 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 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 (hence our T_E too) have quantifier-free fragments decidable for satisfiability. The procedure $\mathsf{SAT}_{\mathsf{MULTI}}$ returns SAT if both satisfiability tests are successful.

4.2 Correctness and Completeness

Before proving correctness and completeness, we introduce useful notation. The notation we introduce is aimed at using language expansions (instead of variable assignments) in Tarski semantics. The formalism of language expansion is adopted in standard mathematical logic textbooks [40] and is a default machinery in all model-theoretic literature.

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 $\mathcal{M}$, the notation $\mathcal{M} \models \varphi(\tilde{\mathbf{c}})$ means that: (i) we expanded the language with free constants naming $\tilde{c}_1, \ldots, \tilde{c}_n$ (the constant naming $\tilde{c}_i$ is called $\tilde{c}_i$ again for simplicity); (ii) the constant naming $\tilde{c}_i$ is interpreted as $\tilde{c}_i$; (iii) in this expansion of $\mathcal{M}$, we have that $\varphi(\tilde{\mathbf{c}})$ turns out to be true (here, according to our general conventions, $\varphi(\tilde{\mathbf{c}})$ is obtained from $\varphi(\mathbf{c})$ by replacing the variables $\mathbf{c}$ with the names of the $\tilde{\mathbf{c}}$).[4]

Below, we assume that F is the $\exists^*\forall^*$-monic-flat formula

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

the formulas F_1, F_2, F_3, F_I, F_E are as described in the decision procedure $\mathtt{SAT}_{\mathtt{MULTI}}$ of Section 4.1.

Lemma 1 (**Completeness of $\mathsf{SAT}_{\mathsf{MULTI}}$**) *If F is $\mathtt{ARR}^2(T_I, T_E)$-satisfiable, then it is possible to choose the set $\mathcal{T}$ and the Boolean assignment $\mathcal{B}$ so that F_I is T_I-satisfiable and F_E is T_E-satisfiable.*

Proof Let $\mathcal{M}$ be a model of F. We have $\mathcal{M} \models \forall \mathbf{i}. \psi(\mathbf{i}, \mathbf{a(i)}, \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}}))$ 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).[5] We take $\mathcal{T}$ to be the set of types real-

[4] People preferring a formulation of Tarski semantics in terms of assignments may interpret $\mathcal{M} \models \varphi(\tilde{\mathbf{c}})$ as meaning that $\varphi(\mathbf{c})$ is true in $\mathcal{M}$ under the assignment mapping the $\mathbf{c}$ to the $\tilde{\mathbf{c}}$.

[5] Notice that this type realization notion is relative to the choice of the elements $\tilde{\mathbf{c}}$ assigned to the $\mathbf{c}$.

ized in $\mathcal{M}$; if $\mathcal{T} = \{M_1, \ldots, M_q\}$, we pick a tuple $\tilde{\mathbf{b}} = \tilde{b}_1, \ldots, \tilde{b}_q$ from $\texttt{INDEX}^{\mathcal{M}}$ realizing them. By assigning precisely this tuple to the variables $\mathbf{b}$ of F_1, we get[6]

$$\mathcal{M} \models \forall x. \left(\bigvee_{j=1}^{q} \bigwedge_{L \in M_j} L(x, \tilde{\mathbf{c}}) \right) \wedge \bigwedge_{j=1}^{q} \bigwedge_{L \in M_j} L(\tilde{b}_j, \tilde{\mathbf{c}}) \wedge \bigwedge_{\sigma : \mathbf{i} \to \tilde{\mathbf{b}}} \psi(\mathbf{i}\sigma, \mathbf{a}(\mathbf{i}\sigma), \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}}))$$

(this formula is F_1 without the outermost existential quantifiers and with **c, b** replaced by - the names of - $\tilde{\mathbf{c}}, \tilde{\mathbf{b}}$). If we furthermore let the tuple $\tilde{\mathbf{e}}$ be the tuple of the elements denoted by the terms $\mathbf{a}[\tilde{\mathbf{c}}] * \mathbf{a}[\tilde{\mathbf{b}}]$, we get

$$\mathcal{M} \models \forall x. \left(\bigvee_{j=1}^{q} \bigwedge_{L \in M_j} L(x, \tilde{\mathbf{c}}) \right) \wedge \bigwedge_{j=1}^{q} \bigwedge_{L \in M_j} L(\tilde{b}_j, \tilde{\mathbf{c}}) \wedge \bar{\psi}(\tilde{\mathbf{b}}, \tilde{\mathbf{c}}, \tilde{\mathbf{e}}) \wedge \bigwedge_{\tilde{d}_m, \tilde{d}_n \in \tilde{\mathbf{b}} * \tilde{\mathbf{c}}} \bigwedge_{l=1}^{s} (\tilde{d}_m = \tilde{d}_n \to \tilde{e}_{l,m} = \tilde{e}_{l,n})$$

as well. Now we can get our $\mathcal{B}$ 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 2 (**Soundness of $\mathsf{SAT}_{\mathsf{MULTI}}$**) *If there exist $\mathcal{T} := \{M_1, \ldots, M_q\}$ and $\mathcal{B}$ such that F_I is T_I-satisfiable and F_E is T_E-satisfiable, then F is $\texttt{ARR}^2(T_I, T_E)$-satisfiable.*

Proof Suppose we are given a set of types $T = \{M_1, \ldots, M_q\}$ and a Boolean assignment $\mathcal{B}$ such that there exists two models $\mathcal{M}_I, \mathcal{M}_E$ of T_I, 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 satisfied in $\mathcal{M}_I$, it follows that there are elements $\tilde{\mathbf{c}}, \tilde{\mathbf{b}}$ from $\texttt{INDEX}^{\mathcal{M}_I}$ such that

$$\mathcal{M}_I \models \forall x. \left(\bigvee_{j=1}^{q} \bigwedge_{L \in M_j} L(x, \tilde{\mathbf{c}}) \right) \wedge \bigwedge_{j=1}^{q} \bigwedge_{L \in M_j} L\left(\tilde{b}_j, \tilde{\mathbf{c}}\right) \wedge \tilde{\psi}_I\left(\tilde{\mathbf{b}}, \tilde{\mathbf{c}}\right) . \tag{1}$$

[6] In particular, $\mathcal{M} \models \forall x. \left(\bigvee_{j=1}^{q} \bigwedge_{L \in M_j} L(x, \tilde{\mathbf{c}}) \right)$ says that at most $M_1, \ldots, M_q$ are realized and $\mathcal{M} \models \bigwedge_{j=1}^{q} \bigwedge_{L \in M_j} L\left(\tilde{b}_j, \tilde{\mathbf{c}}\right)$ says that in fact $M_1, \ldots, M_q$ are realized $\left(\text{by } \tilde{b}_1, \ldots, \tilde{b}_q, \text{respectively}\right)$.

In particular,

$$\mathcal{M}_I \models \bigwedge_{L \in M_j} L\left(\tilde{b}_j, \tilde{\mathbf{c}}\right)$$

holds for every $M_j \in \mathcal{T}$. Thus, each $M_j \in \mathcal{T}$ is associated with an element $\tilde{b}_j \in \texttt{INDEX}^{\mathcal{M}_I}$ that realizes it, while

$$\mathcal{M}_I \models \forall x. \left(\bigvee_{j=1}^{q} \bigwedge_{L \in M_j} L(x, \tilde{\mathbf{c}}) \right) \tag{2}$$

implies that every $\tilde{z} \in \texttt{INDEX}^{\mathcal{M}_I}$ realizes some $M_j \in \mathcal{T}$ (see the proof of the previous Lemma for the definition of type realization). We introduce the following notation: given two elements $\tilde{z}_1, \tilde{z}_2 \in \texttt{INDEX}^{\mathcal{M}_I}$, $\tilde{z}_1 \approx \tilde{z}_2$ holds iff $\tilde{z}_1$ and $\tilde{z}_2$ realize the same type. Thus, for every $\tilde{z} \in \texttt{INDEX}^{\mathcal{M}_I}$ 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 $\tilde{\mathbf{e}} \in \texttt{ELEM}^{\mathcal{M}_E}$ such that (once they are used to interpret the constants $\mathbf{e}$) we have

$$\mathcal{M}_E \models \bar{\psi}_E(\tilde{\mathbf{e}}) \,. \tag{3}$$

To get a model $\mathcal{M}$ for $\texttt{ARR}^2(T_I, T_E)$ we need only to interpret the function symbols $\mathbf{a} = a_1, \ldots, a_s$ as functions from $\texttt{INDEX}^{\mathcal{M}_I}$ into $\texttt{ELEM}^{\mathcal{M}_E}$. Before doing that, let us observe that, because of our choice of $\mathcal{B}$, 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

$$\mathcal{M} \models \bar{\psi}(\tilde{\mathbf{b}}, \tilde{\mathbf{c}}, \tilde{\mathbf{e}}) \wedge \bigwedge_{\tilde{d}_m, \tilde{d}_n \in \tilde{\mathbf{b}} * \tilde{\mathbf{c}}} \bigwedge_{l=1}^{s} \left(\tilde{d}_m = \tilde{d}_n \rightarrow \tilde{e}_{l,m} = \tilde{e}_{l,n} \right) \,. \tag{4}$$

For every $l = 1, \ldots, s$ and for every $\tilde{d}_m \in \tilde{\mathbf{b}} * \tilde{\mathbf{c}}$ we put

$$a_l^{\mathcal{M}}\left(\tilde{d}_m\right) := \tilde{e}_{l,m} \,. \tag{5}$$

By (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

$$a_l^{\mathcal{M}}(\tilde{z}) := a_l^{\mathcal{M}}(\tilde{b}_j) \,. \tag{6}$$

We *claim* that we have, for every $\tilde{z}_1, \tilde{z}_2 \in \texttt{INDEX}^{\mathcal{M}_I}$

$$\tilde{z}_1 \approx \tilde{z}_2 \quad \Rightarrow \quad a_l^{\mathcal{M}}(\tilde{z}_1) = a_l^{\mathcal{M}}(\tilde{z}_2) \,. \tag{7}$$

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 (4) and (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

$$\mathcal{M} \models \forall \mathbf{i}. \psi(\mathbf{i}, \mathbf{a(i)}, \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}})) \,. \tag{8}$$

First notice that, by (5),(4) and by the definition of $\bar{\psi}$(**b, c,e**) (see STEP III of the procedure $\mathsf{SAT}_{\mathsf{MULTI}}$), we have[7]

$$\mathcal{M} \models \bigwedge_{\sigma:\mathbf{i}\to\tilde{\mathbf{b}}} \psi(\mathbf{i}\sigma, \mathbf{a}(\mathbf{i}\sigma), \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}})) \,. \tag{9}$$

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 $\mathtt{INDEX}^{\mathcal{M}}$,[8] we have, for every atom A(**i, a(i), c, a(c)**) occurring in ψ(**i, a(i), c, a(c)**)

$$\mathcal{M} \models A(\tilde{\mathbf{z}}, \mathbf{a}(\tilde{\mathbf{z}}), \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}})) \leftrightarrow A(\tilde{\mathbf{z}}\tau, \mathbf{a}(\tilde{\mathbf{z}}\tau), \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}})) \tag{10}$$

(then (8) follows from (9) and (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 $\mathbf{i} \mapsto \tilde{\mathbf{z}} \mapsto \tilde{z}\mathbf{a}\tau$). In turn, (10) is a special case of the following more general fact: if $\tilde{z}$ and $\tilde{z}'$ have length n and we have $\tilde{z}_i \approx \tilde{z}'_i$ (for every $i = 1, \ldots, n$), then

$$\mathcal{M} \models A(\tilde{\mathbf{z}}, \mathbf{a}(\tilde{\mathbf{z}}), \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}})) \leftrightarrow A(\mathbf{z}', \mathbf{a}(\mathbf{z}'), \tilde{\mathbf{c}}, \mathbf{a}(\tilde{\mathbf{c}})) \tag{11}$$

for every atom A occurring in ψ. However, (11) holds for ELEM atoms thanks to (7) and for INDEX atoms due to the fact that $\tilde{z}_i, \tilde{z}'_i$ realize the same type and the input formula $F := \exists\mathbf{c}\,\forall\mathbf{i}.\psi$(**i, a(i), c, a(c)**) is monic. □

4.3 Complexity Analysis

Theorem 2 applies to $\mathtt{ARR}^2(\mathbb{P}, \mathbb{P})$ because $\mathbb{P}$ admits quantifier elimination. For this theory, we can determine complexity upper and lower bounds:

Theorem 3 *$ARR^2(\mathbb{P}, \mathbb{P})$-satisfiability of $\exists^*\forall^*$-monic-flat sentences is* NEXPTIME*-complete.*

Proof The proof is split into the two lemmas below, giving the lower and upper bound required. □

Lemma 3 (Lower Bound) *$ARR^2(\mathbb{P}, \mathbb{P})$-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 $i < n$;

[7] The elements $\tilde{\mathbf{b}}$ are in bijective correspondence to the variables **b**, hence we can freely suppose that the maps σ indexing the big conjunct of (9) have codomain $\tilde{\mathbf{b}}$.

[8] Recall that n is the length of the tuple **i**. Here $\tilde{\mathbf{z}}$ ranges over all possible tuples of elements that can be assigned to the tuple of variables **i**.

where $\oplus_{2^{n+1}}$ denotes addition modulo 2^{n+1}.

It is well-known [13, 32] 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 $\mathtt{ARR}^2(\mathbb{P}, \mathbb{P})$.

Let us associate (in an injective way) with every element $d \in D$ a numeral (we call this numeral again d for simplicity);[9] 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 coprime 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 [38] - see also [7], saying that the N-th prime number is less than $N \log N + 2N \log \log N$, for all $N > 3$).[10]

We say a natural number i *represents* the point of coordinates $(x, y) \in \left[0, 2^{n+1} - 1\right] \times \left[0, 2^{n+1} - 1\right]$ 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

$$\left(\bigwedge_{k=0}^{n}\left(\neg D_{p_k}(i) \wedge D_{p_k}(i')\right)\right) \vee \bigvee_{k=0}^{n}\left(\bigwedge_{l>k}(D_{p_l}(i) \leftrightarrow D_{p_l}(i')) \wedge \right.$$

$$\left. \wedge D_{p_k}(i) \wedge \neg D_{p_k}(i') \wedge \bigwedge_{l<k}(\neg D_{p_l}(i) \wedge D_{p_l}(i'))\right)$$

- $V_I(i, i')$ stands for the conjunction of $\bigwedge_{k=0}^{n}(D_{p_k}(i) \leftrightarrow D_{p_k}(i'))$ with

$$\left(\bigwedge_{k=0}^{n}(\neg D_{q_k}(i) \wedge D_{q_k}(i'))\right) \vee \bigvee_{k=0}^{n}\left(\bigwedge_{l>k}(D_{q_l}(i) \leftrightarrow D_{q_l}(i')) \wedge \right.$$

$$\left. \wedge D_{q_k}(i) \wedge \neg D_{q_k}(i') \wedge \bigwedge_{l<k}(\neg D_{q_l}(i) \wedge D_{q_l}(i'))\right)$$

[9] 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 $\mathtt{ARR}^2(\mathbb{P}, Eq)$, where Eq is the pure identity theory in a language containing infinitely many constants constrained to be distinct.

[10] 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 coprime, 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 $h(n + 1) \leq 1 + \prod_{m \leq n} h(m) \leq 1 + n \cdot n! \leq (n + 1)!$.

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 $y' = y \oplus_{2^{n+1}} 1$.

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):

$$
\begin{aligned}
&\bigwedge_{k=0}^{n-1} \forall i \, (P_{k,0}(i) \rightarrow a[i] = d_k) \wedge \\
&\wedge \; \forall i_1 \forall i_2 \, (H_I(i_1, i_2) \rightarrow H_E(a[i_1], a[i_2])) \wedge \\
&\wedge \; \forall i_1 \forall i_2 \, (V_I(i_1, i_2) \rightarrow V_E(a[i_1], a[i_2])) \; .
\end{aligned}
$$

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 4 (**Upper Bound**) $ARR^2(\mathbb{P}, \mathbb{P})$*-satisfiability of* $\exists^*\forall^*$*-monic-flat sentences is in* NExpTime.

Proof 1 To show the matching upper bound, it is sufficient to inspect our decision algorithm $\mathtt{SAT}_{\mathtt{MULTI}}$. Clearly, Step I introduces an exponential guess; the formulas 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 $\mathcal{B}$ can be seen as a set of exponentially many literals). It is well-known that $\mathbb{P}$-satisfiability of quantifier-free formulas is in NP (see the historical references in [36] for the origins of this result), so that satisfiability of F_E also takes non deterministic exponential time. We only have to discuss $\mathbb{P}$-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

$$
\neg \exists x \neg \left(\bigvee_{j=1}^{q} \bigwedge_{L \in M_j} L(x, \mathbf{c}) \right) \tag{12}
$$

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 Presburger arithmetic. We need the following lemma from [36] (Theorem 1, p.327):

Lemma 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 (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 (12). Now (quoting from [36], 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.

5 A Decidability Result for the Reachability Analysis of Flat Array Programs

Based on the decidability results described in the previous section, we can now achieve important decidability results in the context of reachability analysis for programs handling arrays of unbounded length. As a reference theory, we shall use $\mathtt{ARR}^1(\mathbb{P}^+)$ or $\mathtt{ARR}^2(\mathbb{P}^+, \mathbb{P}^+)$, where $\mathbb{P}^+$ is $\mathbb{P}$ enriched with free constant symbols and with *definable* predicate and function symbols. We do not enter into more details concerning what a definable symbol is (see, e.g., [40]), we just underline that definable symbols are nothing but useful macros that can be used to formalize case-defined functions and SMT-LIB commands like if-then-else. The addition of definable symbols does not compromise quantifier elimination, hence decidability of $\mathbb{P}^+$. Below, we let $\mathcal{T}$ be $\mathtt{ARR}^1(\mathbb{P}^+)$ or $\mathtt{ARR}^2(\mathbb{P}^+, \mathbb{P}^+)$.

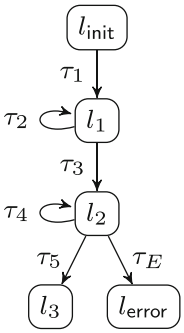
Henceforth $\mathbf{v}$ will denote, in the following, the variables of the programs we will analyze. Formally, $\mathbf{v} = \mathbf{a}, \mathbf{c}$ where, according to our conventions, $\mathbf{a}$ is a tuple of array variables (modeled as free unary function symbols of $\mathcal{T}$ in our framework) and $\mathbf{c}$ a tuple of scalar variables; the latter can be considered as variables in the logical sense - in $\mathtt{ARR}^2(\mathbb{P}^+, \mathbb{P}^+)$ we can conveniently model them either as variables of sort INDEX or as free constants of sort ELEM.

A *state-formula* is a formula $\alpha(\mathbf{v})$ of $\mathcal{T}$ representing a (possibly infinite) set of configurations of the program under analysis. A *transition formula* is a formula of $\mathcal{T}$ of the kind $\tau(\mathbf{v}, \mathbf{v}')$ where $\mathbf{v}'$ is obtained from copying the variables in $\mathbf{v}$ and adding a prime to each of them. For the purpose of this work, programs will be represented by their control-flow automaton.

Definition 1 (Programs) Given a set of variables $\mathbf{v}$, a *program* is a triple $\mathcal{P} = (L, \Lambda, E)$, where (i) $L = \{l_1, \ldots, l_n\}$ is a set of *program locations* among which we distinguish an initial location l_{init} and an error location l_{error}; (ii) Λ is a finite set of transition formulas $\{\tau_1(\mathbf{v}, \mathbf{v}'), \ldots, \tau_r(\mathbf{v}, \mathbf{v}')\}$ and (iii) $E \subseteq L \times \Lambda \times L$ is a set of *actions*.

```
  procedure initEven ( a[N] , v ) :
l1    for (i = 0; i < N; i = i + 2)  a[i] = v;
l2    for (i = 0; i < N; i = i + 2)  assert(a[i] = v);
```

(a)



(b)

Fig. 1 The initEven procedure (a) and its control-flow graph (b)

We indicate by $src, \mathcal{L}, tgt$ the three projection functions on E; that is, for $e = (l_i, \tau_j, l_k) \in E$, we have $src(e) = l_i$ (this is called the 'source' location of e), $\mathcal{L}(e) = \tau_j$ (this is called the 'label' of e) and $tgt(e) = l_k$ (this is called the 'target' location of e).

Example 2 Consider the procedure initEven in Fig. 1. For this procedure, $\mathbf{a} = a, \mathbf{c} = i, v$. N is a constant of the background theory. Λ is the set of formulas (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 Fig. 1b, where $L = \{l_{\text{init}}, l_1, l_2, l_3, l_{\text{error}}\}$.

Definition 2 (**Program paths**) A *program path* (in short, *path*) of $\mathcal{P} = (L, \Lambda, E)$ is a sequence $\rho \in E^n$, i.e., $\rho = e_1, e_2, \ldots, e_n$, such that for every $e_i, e_{i+1}, tgt(e_i) = src(e_{i+1})$. We denote with $|\rho|$ the length of the path. An *error path* is a path ρ with $src(e_1) = l_{\text{init}}$ and $tgt(e_{|\rho|}) = l_{\text{error}}$. 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 $\tau_{i_j}\left(\mathbf{v}^{(j-1)}, \mathbf{v}^{(j)}\right)$, with $\mathcal{L}(e_j) = \tau_{i_j}$ (the notation $\tau_{i_j}\left(\mathbf{v}^{(j-1)}, \mathbf{v}^{(j)}\right)$ 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 $\mathcal{P}$ is to detect if $\mathcal{P}$ admits a feasible error path. Proving the safety of $\mathcal{P}$, therefore, means solving the reachability problem for $\mathcal{P}$. This problem, given well known limiting results, is not decidable for an arbitrary program $\mathcal{P}$. The consequence is that, in general, reachability analysis is sound, but not complete,

and its incompleteness manifests itself in (possible) divergence of the verification algorithm (see, e.g., [1, 3]).

To gain decidability, we must first impose restrictions on the shape of the transition formulas, for instance we can constrain the analysis to formulas falling within decidable classes like those we analyzed in the previous section. 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 [15, 17, 22] for integer variable programs, but since we handle array variables we need specific restrictions for acceleration. Our result for the decidability of the safety of annotated ARR ay programs builds upon the results presented in Sections 3 and 4 and the acceleration procedure presented in [4].

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 $\mathsf{flat}^0 - program$.

Definition 3 (**flat^0-program**) A program $\mathcal{P}$ is a $\mathsf{flat}^0 - program$ if for every path $\rho = e_1, \ldots, e_n$ of $\mathcal{P}$ it holds that for every $j < k$ $(j, k \in \{1, \ldots, n\})$, if $src(e_j) = tgt(e_k)$ then $e_j = e_{j+1} = \cdots = e_k$.

We now turn our attention to transition formulas. Acceleration is a well-known formalism in the area of model-checking. It has been integrated in several frameworks and constitutes a fundamental technology for the scalability and efficiency of modern model checkers (e.g., [10]). 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. 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. By definition, the acceleration of a transition $\tau(\mathbf{v}, \mathbf{v}')$ is the union of the n-th compositions of τ with itself, i.e. it is $\tau^+ := \bigvee_{n>0} \tau^n$, where

$$\tau^1(\mathbf{v}, \mathbf{v}') := \tau(\mathbf{v}, \mathbf{v}'), \qquad \tau^{n+1}(\mathbf{v}, \mathbf{v}') := \exists \mathbf{v}''.(\tau(\mathbf{v}, \mathbf{v}'') \wedge \tau^n(\mathbf{v}'', \mathbf{v}')) \,.$$

τ^+ can be practically exploited only if there exists a formula $\varphi(\mathbf{v}, \mathbf{v}')$ equivalent, in the models of the considered background theory, to $\bigvee_{n>0} \tau^n$.

Based on this observation, 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, as we did for results in Sections 3 and 4, modular and general. We will show instances of this result in Section 6. Notationally, let us modify the projection function $\mathcal{L}$ by putting $\mathcal{L}^+(e) := \mathcal{L}(e)^+$ if $src(e) = tgt(e)$ and $\mathcal{L}^+(e) := \mathcal{L}(e)$ otherwise, where $\mathcal{L}(e)^+$ denotes the acceleration of the transition labeling the edge e.

Theorem 4 *Let $\mathcal{F}$ be a class of formulas decidable for $\mathcal{T}$-satisfiability. The unbounded reachability problem for a $\mathsf{flat}^0 - program$ $\mathcal{P}$ is decidable if (i) $\mathcal{L}$ is closed under conjunctions and (ii) for each $e \in E$ one can compute $\alpha(\boldsymbol{v}, \boldsymbol{v}') \in \mathcal{L}$ such that $\mathcal{T} \models \mathcal{L}^+(e) \leftrightarrow \alpha(\boldsymbol{v}, \boldsymbol{v}')$.*

Proof Let $\rho = e_1, \ldots, e_n$ be an error path of $\mathcal{P}$; when testing its feasibility, according to Definition 3, 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 2.[11] Thus $\mathcal{P}$ is unsafe iff, for some path $e_1, \ldots, e_n$ whose edges are all distinct, the formula

$$\mathcal{L}^+(e_1)^{(1)} \wedge \cdots \wedge \mathcal{L}^+(e_n)^{(n)} \tag{13}$$

is $\mathcal{T}$-satisfiable. Since the involved paths are finitely many and $\mathcal{T}$-satisfiability of formulas like (13) is decidable, the safety of $\mathcal{P}$ can be decided. □

5.1 A Class of Array Programs with Decidable Reachability Problem

We now produce a class of programs with arrays – we call it $\mathsf{simple}^0_{\mathcal{A}}$progams – for which requirements of Theorem 4 are met. The class of $\mathsf{simple}^0_{\mathcal{A}}$-progams contains non recursive programs implementing searching, copying, comparing, initializing, replacing and testing procedures. As an example, the initEven program reported in Fig. 1 is a $\mathsf{simple}^0_{\mathcal{A}}$-progam. Formally, a $\mathsf{simple}^0_{\mathcal{A}}$ – $progam\mathcal{P} = (L, \Lambda, E)$ 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 1 or Theorem 3; (ii) if $e \in E$ is a self-loop, then $\mathcal{L}(e)$ is a simple_k-assignment.

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 4 (simple_k-assignment) Let $k \neq 0$; a simple_k – $assignment$ is a transition $\tau(\mathbf{v}, \mathbf{v}')$ of the kind

$$\phi_L(\mathbf{c}, \mathbf{a}(d)) \wedge d' = d + k \wedge \mathbf{d}' = \mathbf{d} \wedge \mathbf{a}' = \lambda j.\mathsf{if}\ (j = d)\ \mathsf{then}\ \mathbf{t}(\mathbf{c}, \mathbf{a}(d))\ \mathsf{else}\ \mathbf{a}(j)$$

where (i) $\mathbf{c} = d, \mathbf{d}$ and (ii) the quantifier-free formula $\phi_L(\mathbf{c}, \mathbf{a}(d))$ and the terms $\mathbf{t}(\mathbf{c}, \mathbf{a}(d))$ are flat.

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, $\mathbf{t}(\mathbf{c}, \mathbf{a}(d))$ stands for an s-tuple of terms $t_1(\mathbf{c}, \mathbf{a}(d), \ldots, t_s(\mathbf{c}, \mathbf{a}(d))$. 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[12]

$$\bigwedge_{h=1}^{s} \forall j. \begin{pmatrix} (j = d \wedge a'_h(j) = t_h(\mathbf{c}, \mathbf{a}(d))) \vee \\ \vee\ (j \neq d \wedge a'_h(j) = a_h(j)) \end{pmatrix} \tag{14}$$

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 use different scanners for the different arrays (one scanner for each of them) with different increments; such generalization is easy and left to the reader (we prefer not to formally introduce it in order not to complicate the notation).

[11] Notice that by these replacements we can represent in one shot infinitely many paths, namely those executing self-loops any given number of times.

[12] Using McCarthy [33] update notation, we can write it as $\bigwedge_h a'_h = upd(a_h, d, t_h(\mathbf{c}, \mathbf{a}(d)))$.

The following Lemma (which is an instance of a more general result from [4]) gives the template for the accelerated counterpart of a simple_k-assignment.

Lemma 6 *Let $\tau(\boldsymbol{v}, \boldsymbol{v}')$ be a $\mathsf{simple}_k - assignment$ like in Definition 4. Then $\tau^+(\boldsymbol{v}, \boldsymbol{v}')$ is $\mathcal{T}$-equivalent to the formula*

$$\exists y > 0 \left(\begin{array}{l} \forall z.\, (d \leq z < d + k \cdot y \wedge D_k(z - d) \rightarrow \ \phi_L(z, \boldsymbol{d}, \boldsymbol{a}(z))) \ \wedge \\ \boldsymbol{a}' = \lambda j.\boldsymbol{U}(j, y, \boldsymbol{v}) \ \wedge \ d' = d + k \cdot y \ \wedge \ \boldsymbol{d}' = \boldsymbol{d} \end{array} \right) \tag{15}$$

where the definable functions $U_h(j, y, \boldsymbol{v})$, $1 \leq h \leq |\boldsymbol{a}|$, of the tuple of functions $\boldsymbol{U}$ are

$$\mathit{if}\ (d \leq j < d + k \cdot y \ \wedge D_k(j - d))\ \mathit{then}\ t_h(j, \boldsymbol{d}, \boldsymbol{a}(j))\ \mathit{else}\ a_h(j)\ .$$

Proof It is sufficient to check by induction on $y \geq 1$ that if we execute y-times the $\mathsf{simple}_k - assignment$ of Definition 4, we get

$$\begin{array}{l} \forall z.\, (d \leq z < d + k \cdot y \wedge D_k(z - d) \rightarrow \ \phi_L(z, \mathbf{d}, \mathbf{a}(z))) \ \wedge \\ \wedge \ \mathbf{a}' = \lambda j.\mathbf{U}(j, y, \mathbf{v}) \ \wedge \ d' = d + k \cdot y \ \wedge \ \mathbf{d}' = \mathbf{d} \end{array}$$

which means that the accelerated assignment is described by (15). □

Example 3 Consider transition τ_2 from the formalization of our running example of Fig. 1. The acceleration τ_2^+ of such formula is (we omit identical updates)

$$\exists y > 0.\, \left(\begin{array}{l} \forall z.(i \leq z < i + 2y \wedge D_2(z - i) \rightarrow z < N) \wedge i' = i + 2y \wedge \\ a' = \lambda j.\, (\mathsf{if}\ \ (i \leq j < 2y + i \wedge D_2(j - i))\ \ \mathsf{then}\ v\ \mathsf{else}\ a(j)) \end{array} \right)$$

We can now formally show that the reachability problem for $\mathsf{simple}^0_{\mathcal{A}}$-progams is decidable, by instantiating Theorem 4 with the results obtained so far.

Theorem 5 *The unbounded reachability problem for $\mathsf{simple}^0_{\mathcal{A}}$ progams 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 1 or Theorem 3 are closed under conjunctions. Since the acceleration of a $\mathsf{simple}_k - assignment$ fits inside these classes (just eliminate definitions via λ-abstractions by using universal quantifiers, like in (14)), Theorem 4 applies. □

6 Experimental Observations

SMT-solvers constitute nowadays one of the fundamental components in tools dealing with the formal verification of software systems. Given the growing demand for solvers supporting quantified fragments of theory of interests from a practical perspective, different SMT-solvers started implementing procedures for handling quantified formulas. As an example, cvc4 [8], VERIT [14] and Z3 [18] are SMT-solvers, according to [21], able to deal with sentences over the theory of arrays over linear arithmetic (constraining both indices and values).

Nowadays, the vast majority of the software model-checkers exploit abstraction techniques in order to check the safety of a program, as witnessed by the report of the last international competition on software verification (SV-COMP, [11]). Checking the safety of (some) $\mathsf{simple}^0_{\mathcal{A}}$-progams with abstraction techniques can be really challenging. As an example, proving the safety of the procedure initEven of Fig. 1 (i.e., generating a safe inductive invariant for it) requires a reasoning schema able to realize the need of a divisibility predicate. This is a non trivial task, and even highly engineered tools, e.g., those described in [12], are not able to deal with this problem. Implementing the procedure identified in the proof of Theorem 4 as a preprocessing step in a software model-checker (partially[13]) overcomes this problem, as $\mathsf{simple}^0_{\mathcal{A}}$-progams will not be analyzed at all by the abstraction module.

In our case, we implemented the decision procedure described in the proof of Theorem 4 in the tool BOOSTER [5] as an additional analysis technique targeting the verification of $\mathsf{simple}^0_{\mathcal{A}}$-progams. This section reports on our experimental findings. Indeed, the decision procedures presented in this paper have not been implemented in any SMT-solver. It is natural to ask, therefore, whether there is a real need for them in practice or if the available solvers are already able to cope with those Flat Array Properties arising from pragmatic solutions for the analysis of computer systems.

BOOSTER has been built over Z3. Z3 has several ways for dealing with quantifiers: it offers an implementation of the *matching modulo equalities* (E-matching) solution [19], the decision procedures for arrays described in [16] and a more advanced instantiation procedures, *Model Based Quantifier Instantiation* (MBQI), described in [24]. While the first two procedures are generally not enough for checking Flat Array Properties (the first is not automatic since an "instantiation pattern" has to be suggested while the second works on formulas produced by a grammar not matching Flat Array Properties, see the discussion in the next section), the MBQI procedure works pretty well with our Flat Array Properties, even though fails on a few of them. More precisely, the solver detects fastly the unsatisfiability of inconsistent formulas and the only slow-downs are detected for a few satisfiable instances. The other SMT-solvers implementing features for dealing with quantified formulas fail as well with such instances.

We report below a typical example example of a Flat Array Property on which we observed the failure of the SMT-solvers.[14] The failure is due to the presence of constraints (like divisibility predicates) requiring instantiation strategies that, in order to be complete, must be rather sophisticated.

6.1 A Concrete Example of a Flat Array Property

As an example of Flat Array Property out of reach for the available SMT-solvers, consider the mergeInterleave procedure taken from [20] and reported in Fig. 2a. For this procedure,

[13] This solution fully works only if the input program is a $\mathsf{simple}^0_{\mathcal{A}}$-progams. Otherwise, however, acceleration procedures for arrays can be adopted as well inside a abstraction-refinement loop, as described in [4].

[14] The versions of the solvers we tested are the following: cvc4, version 1.4-prerelease. VERIT, version 201310d. Z3, version 4.3.1.

$\mathbf{a} = a, b, r$, $\mathbf{c} = i, k$. N is a constant of the background theory. Λ is the set of formulas (we omit identical updates and the transitions not leading to error locations):

$$\begin{aligned}
\tau_1 &:= i' = 0\\
\tau_2 &:= i < N \wedge r' = \lambda j.\mathsf{if}\ (j = i)\ \mathsf{then}\ a(j)\ \mathsf{else}\ r(j) \wedge i' = i + 2\\
\tau_3 &:= i \geq N \wedge i' = 1\\
\tau_4 &:= i < N \wedge r' = \lambda j.\mathsf{if}\ (j = i)\ \mathsf{then}\ b(j)\ \mathsf{else}\ r(j) \wedge i' = i + 2\\
\tau_5 &:= i \geq N\\
\tau_{E_1} &:= k \geq 0 \wedge k < N \wedge k \equiv_2 0 \wedge r[k] \neq b[k]\\
\tau_{E_2} &:= k \geq 0 \wedge k < N \wedge k \equiv_2 1 \wedge r[k] \neq a[k]
\end{aligned}$$

The procedure mergeInterleave can be formalized as the control-flow graph depicted in Fig. 2b (as before, we are not reporting edges of the control-flow graph that are not considered for checking the safety of the procedure), where $L = \{l_{\mathsf{init}}, l_1, l_2, l_3, l_{\mathsf{error}}\}$.

Transitions τ_2 and τ_4 are simple_k-assignments. Their accelerations are (omitting identical updates):

$$\tau_2^+ := \exists y. \begin{pmatrix} y > 0 \wedge \forall j.((i \leq j < i + 2y \wedge D_2(j - i)) \rightarrow j < N) \wedge \\ i' = i + 2y \wedge r' = \lambda j.\mathsf{if}\ (i \leq j < 2y + i \wedge D_2(j - i))\ \mathsf{then}\ a(j)\ \mathsf{else}\ r(j) \end{pmatrix}$$

and

$$\tau_4^+ := \exists y. \begin{pmatrix} y > 0 \wedge \forall j.((i \leq j < i + 2y \wedge D_2(j - i)) \rightarrow j < N) \wedge \\ i' = i + 2y \wedge r' = \lambda j.\mathsf{if}\ (i \leq j < 2y + i \wedge D_2(j - i))\ \mathsf{then}\ b(j)\ \mathsf{else}\ r(j) \end{pmatrix}$$

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:

$$\begin{aligned}
&i_1 = 0 \wedge \forall j.r_1(j) = r_0(j) \wedge\\
&\exists y_1. \begin{pmatrix} y_1 > 0 \wedge i_2 = i_1 + 2y_1 \wedge \\ \forall j.((i_1 \leq j < i_1 + 2y_1 \wedge D_2(j - i_1)) \rightarrow j < N) \wedge \\ \forall j.(r_2(j) = \mathsf{if}\ (i_1 \leq j < 2y_1 + i_1 \wedge D_2(j - i_1))\ \mathsf{then}\ a(j)\ \mathsf{else}\ r_1(j)) \end{pmatrix} \wedge\\
&i_2 \geq N \wedge i_3 = 1 \wedge \forall j.(r_3(j) = r_2(j)) \wedge\\
&\exists y_3. \begin{pmatrix} y_3 > 0 \wedge i_4 = i_3 + 2y_3 \wedge \\ \forall j.((i_3 \leq j < i_3 + 2y_3 \wedge D_2(j - i_3)) \rightarrow j < N) \wedge \\ \forall j.(r_4(j) = \mathsf{if}\ (i_3 \leq j < 2y_3 + i_3 \wedge D_2(j - i_3))\ \mathsf{then}\ b(j)\ \mathsf{else}\ r_3(j)) \end{pmatrix} \wedge\\
&i_4 \geq N \wedge i_5 = i_4 \wedge \forall j.(r_5(j) = r_4(j)) \wedge\\
&0 \leq k \wedge k < N \wedge D_2(k) \wedge r_5(k) \neq b(k) \wedge\\
&i_6 = i_5 \wedge \forall j.(r_6(j) = r_5(j))
\end{aligned}$$

7 Related Work

The modular nature of our solution makes our contributions orthogonal with respect to the state of the art: we can enrich $\mathbb{P}$ with various definable or even not definable symbols [39] and get from our Theorems 1, 2 decidable classes which are far from the scope of existing

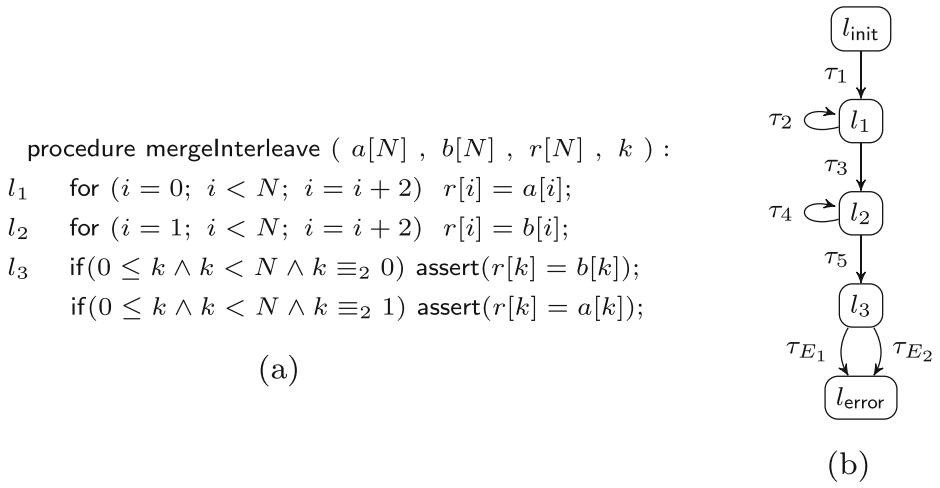


Fig. 2 The mergeInterleave procedure (a) and its control-flow graph (b)

results. Given the parameterized nature of our results, there is some similarity with [30], although (contrary to [30]) we consider purely syntactically specified classes of formulas. It is interesting to notice that also the special cases of the decidable classes covered by Corollary 1 and Theorem 3 are orthogonal to the results from the literature. To this aim, we make a closer comparison with [16, 27].

The two fragments considered in [16, 27] are characterized by rather restrictive syntactic constraints. In [27] it is considered a subclass of the $\exists^*\forall$-fragment of $\mathtt{ARR}^1(T)$ called SIL, *Single Index Logic*. In this class, formulas are built according to a grammar allowing (i) as atoms occurring in universally quantified subformulas only difference logic constraints and some equations modulo a fixed integer and (ii) as universally quantified subformulas only formulas 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 formulas, due to the dereference applied to increment terms $\mathbf{i}+\mathbf{k}$ in the consequents of the above universally quantified implications. Similar restrictions are in [28].

The Array Property Fragment described in [16] is basically a subclass of the $\exists^*\forall^*$-fragment of $\mathtt{ARR}^2(\mathbb{P}, \mathbb{P})$; however universally quantified subformulas are constrained to be of the kind $\forall\mathbf{i}.\phi(\mathbf{i}) \rightarrow \psi(\mathbf{a}(\mathbf{i}))$, where in addition the $\mathtt{INDEX}$ 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 formulas 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 1, while the complexity of the decision procedure proposed in [27] is unknown. On the other side, both $\mathsf{SAT}_{\mathsf{MULTI}}$ and the decision procedure described in [16] run in NExpTime. The decision procedure in [16] 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 formulas are also partially different, in spirit, from those presented so far in the SMT community. While the vast majority of SMT-Solvers

address the problem of checking the satisfiability of quantified formulas via instantiation (see, e.g., [16, 19, 24, 37]), 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 4.1) 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 subgoals is an additional specific feature of our satifiability procedures.

From the point of view of the applications, providing a full decidability result for the unbounded reachability analysis of a class of array programs is what differentiates our work with other contributions like [1, 3, 4].

8 Conclusions and Future Work

In this paper we identified a class of Flat Array Properties, a quantified fragment of theories of arrays, admitting decision procedures. We provided a complexity analysis of our decision procedures. We also showed that the decidability of Flat Array Properties, combined with acceleration results, allows to depict a sound and complete procedure for checking the safety of a class of programs with arrays.

A thorough experimental evaluation of the new procedures presented in this paper is still missing and is deferred to future work. We just make here some observations concerning implementation. First of all, it must be pointed out that the fragments covered by our decision procedures are quite expressive - witness the above mentioned fact that the NEXPTIME lower bound is attained by $\mathsf{SAT}_{\mathsf{MULTI}}$ already by fixing the number of the universally quantified variables. It is not clear, on the other hand, whether this large expressivity is needed in concrete applications (the encoding of the domino problem in Section 4.3 seems to be far from the kind of formulas arising from software problems like those analyzed in Section 5). However, we have seen in Section 6 that there are concrete problems where pure instantiation procedures are insufficient: as it is evident from the analysis of the completeness proof in [16], the rationale behind instantiation procedures like that of [16] is that (up to satisfiability) we limit to models where arrays are 'constants on definable intervals'. This assumption is not appropriate for instance for procedures like initEven or mergeInterleave where arrays are scanned in a non uniform way: it is precisely in these cases that we need to enlarge the set of instantiation terms using extra constants witnessing type realizations. Of course type guessing is one of the ingredients making the algorithm $\mathsf{SAT}_{\mathsf{MULTI}}$ highly problematic. To alleviate the problem, we probably need to have a closer look to the formulae arising in the applications. For instance, array accelerations (15) typically partition the array domain into intervals and thus naturally indicate the right set of types that has to be realized: most information given by a type over a variable z consists in appropriately locating z inside an interval, hence it is obvious that all these types have to be realized (unless intervals are somewhat degenerated). Thus non-determinism seems to be limited if we look at the applications; a similar remark applies to the other source of complexity of $\mathsf{SAT}_{\mathsf{MONO}}$ and $\mathsf{SAT}_{\mathsf{MULTI}}$, namely the need of discharging quantified arithmetic subgoals. For instance, in $\mathsf{SAT}_{\mathsf{MULTI}}$ the quantifier elimination applied to the arithmetic subformula $\forall x.\left(\bigvee_{j=1}^{q}\bigwedge_{L\in M_j} L(x,\mathbf{c})\right)$ introduced in Step II–V computes the (often trivial) side effects of the missed realization of some types (for instance if there is no even z inside an interval $[t,u]$ this is because the interval is degenerated and $t=u$ is odd).

In conclusion, given that the state-of-the-art techniques for handling quantifiers seem to be insufficient, the additional methodologies indicated in this paper, although hardly feasible in their full generality, seem to indicate promising techniques in order to attack intermediate classes of problems suggested by concrete applications.

References

1. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy abstraction with interpolants for arrays. In: LPAR, pp. 46–61 (2012)
2. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: SAFARI: SMT-Based Abstraction for Arrays with Interpolants. In: CAV, pp. 679–685 (2012)
3. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: An extension of lazy abstraction with interpolation for programs with arrays. Formal Methods in System Design **45**(1), 63–109 (2014)
4. Alberti, F., Ghilardi, S., Sharygina, N.: Definability of accelerated relations in a theory of arrays and its applications. In: FroCoS, pp. 23–39 (2013)
5. Alberti, F., Ghilardi, S., Sharygina, N.: Booster: An acceleration-based verification framework for array programs. In: Cassez, F., Raskin, J.-F. (eds.) 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, pp. 18–23, Springer (2014)
6. Alberti, F., Ghilardi, S., Sharygina, N.: Decision procedures for flat array properties. In: TACAS (2014)
7. Bach, E., Shallit, J.: Algorithmic Number Theory. Vol. 1. Foundations of Computing Series. MIT Press (1996)
8. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV, pp. 171–177 (2011)
9. Barrett, C., Stump A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2010)
10. Behrmann, G., Bengtsson, J., David, A., Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL implementation secrets. In: FTRTFT, pp. 3–22 (2002)
11. Beyer, D.: Status report on software verification - (competition summary sv-comp 2014). In: Ábrahám, E., Havelund, K. (eds.) TACAS, volume 8413 of Lecture Notes in Computer Science, pp. 373–388. Springer (2014)
12. Bjørner, N., McMillan, K.L., Rybalchenko, A.: On solving universally quantified horn clauses. In: SAS, pp. 105–125 (2013)
13. Börger, E., Grädel, E., Gurevich, Y.: The Classical Decision Problem. Perspectives in Mathematical Logic. Springer-Verlag, Berlin (1997)
14. Bouton, T., Caminha, D., de Oliveira, B., Déharbe, D., Fontaine, P.: Verit: An open, trustable and efficient smt-solver. In: Schmidt, R.A. (ed.) CADE, volume 5663 of Lecture Notes in Computer Science, pp. 151–156. Springer (2009)
15. Bozga, M., Iosif, R., Lakhnech, Y.: Flat parametric counter automata. Fundamenta Informaticae **91**, 275–303 (2009)
16. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: VMCAI, pp. 427–442 (2006)
17. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and presburger arithmetic. In: CAV, vol. 1427 of LNCS, pp. 268–279. Springer (1998)
18. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS, pp. 337–340 (2008)
19. Detlefs, D.L., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Technical Report HPL-2003-148, HP Labs (2003)
20. Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. weak updates. In: ESOP, pp. 246–266 (2010)
21. Weber, T., Cok, D.R., Stump, A.: The 2013 SMT Evaluation. Available at http://smtcomp.sourceforge.net/2013/report/SMTEVAL-2013.pdf (2013)
22. Finkel, A., Leroux, J.: How to compose Presburger-accelerations: Applications to broadcast protocols. In: FSTTCS, pp. 145–156 (2002)
23. Ganzinger, H.: Shostak light. Automated deduction—CADE-18, vol. 2392 of Lecture Notes in Comput. Sci., pp. 332–346. Springer, Berlin (2002)
24. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: CAV, pp. 306–320 (2009)

25. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. Logical Methods in Computer Science **6**(4) (2010)
26. Ghilardi, S., Ranise, S.: MCMT: A Model Checker Modulo Theories. In: IJCAR, pp. 22–29 (2010)
27. Habermehl, P., Iosif, R., Vojnar, T.: A logic of singly indexed arrays. In: LPAR, pp. 558–573 (2008)
28. Habermehl, P., Iosif, R., Vojnar, T.: What else is decidable about integer arrays? In: FOSSACS (2008)
29. Halpern, J.Y.: Presburger arithmetic with unary predicates is Π_1^1 complete. J. Symbolic Logic **56**(2), 637–642 (1991)
30. Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: On local reasoning in verification. In: TACAS, pp. 265–281. Springer (2008)
31. Jhala, R., McMillan, K.L.: Array Abstractions from Proofs. In: CAV (2007)
32. Lewis, H.B.: Complexity of solvable cases of the decision problem for the predicate calculus. In: 19th Ann. Symp. on Found. of Comp. Sci., pp. 35–47. IEEE (1978)
33. McCarthy, J.: Towards a mathematical science of computation. In: International Federation for Information Processing Congress, pp. 21–28 (1962)
34. McMillan, K.L.: Lazy Abstraction with Interpolants. In: CAV (2006)
35. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic. In: CAV'05, pp. 321–334 (2005)
36. Oppen, D.C.: A superexponential upper bound on the complexity of Presburger arithmetic. J. Comput. Syst. Sci. **16**(3), 323–332 (1978)
37. Reynolds, A., Tinelli, C., Goel, A., Krstic, S., Deters, M., Barrett, C.: Quantifier instantiation techniques for finite model finding in SMT. In: CADE, pp. 377–391 (2013)
38. Rosser, B.: The n-th prime is greater than n log n. Proc. Lond. Math. Soc., II. Ser. **45**, 21–44 (1938)
39. Semënov, A.L.: Logical theories of one-place functions on the set of natural numbers. Izvestiya: Mathematics **22**, 587–618 (1984)
40. Shoenfield, J.R.: Mathematical logic. Association for Symbolic Logic, Urbana, IL. Reprint of the 1973 second printing (2001)
41. Tinelli, C., Zarba, C.G.: Combining nonstably infinite theories. J. Automat. Reason. **34**(3), 209–238 (2005)