A Combination of Rewriting and Constraint Solving for the Quantifier-free Interpolation of Arrays with Integer Difference Constraints

Roberto Bruttomesso[1] Silvio Ghilardi[2] Silvio Ranise[3]

[1] Università della Svizzera Italiana, Lugano, Switzerland
[2] Università degli Studi di Milano, Milan, Italy
[3] FBK (Fondazione Bruno Kessler), Trento, Italy

Abstract. The use of interpolants in model checking is progressively gaining importance. The application of encodings based on the theory of arrays, however, is limited by the impossibility of deriving quantifier-free interpolants in general. To overcome this problem, we have recently proposed a quantifier-free interpolation solver for a natural variant of the theory of arrays. However, arrays are usually combined with fragments of arithmetic over indexes in applications, especially those related to software verification. In this paper, we propose a quantifier-free interpolation solver for the variant of arrays considered in previous work when combined with integer difference logic over indexes.

1 Introduction

Arrays are essential data-structures in computer science. The problem of verifying functional correctness of software and hardware components using symbolic model-checking techniques often boils down to the problem of checking properties over arrays and arithmetic, expressed as quantifier-free first order logic formulæ. Consider for example the following pseudo-code fragment

$$\begin{array}{l}\text{for (int } i = 0 \text{ ; } i \leq n-1 \text{ ; } i = i+1 \text{)}\\ \quad \text{if (} a[i] > a[i+1] \text{)}\\ \quad\quad \text{swap(} a[i], a[i+1] \text{);}\end{array}$$

This loop is used, e.g., in bubble-sort to move the maximum element in the range $[0, n]$ of the array a to position n. It thus satisfies the postcondition

$$\forall i.\ 0 \leq i \leq n-1 \implies a[i] \leq a[n].$$

A possible approach to model-check such property can be established by taking its negation ($\exists i.\ 0 \leq i \leq n-1 \ \wedge\ a[i] > a[n]$), which is an "unsafety condition", and by running a symbolic reachability procedure. State-of-the-art methods for reachability are based on an abstraction-refinement loop, where the refinement phase is handled by means of the computation of interpolants [10].

In order to apply this method it is necessary to provide procedure that computes quantifier-free interpolants for unsatisfiable quantifier-free formulæ in the

theories under consideration. For instance a symbolic encoding of our example above would be naturally defined in the combination of the theory of arrays ($\mathcal{AX}$) and integer difference logic ($\mathcal{IDL}$). However it is known that, already in $\mathcal{AX}$, quantifier-free interpolants cannot be produced in general.

In a recent work [5], we have shown an extension of $\mathcal{AX}$ with a further functional symbol `diff`, called $\mathcal{AX}_{\texttt{diff}}$, in which quantifier-free interpolants can be computed. In this paper, we extend that result by augmenting $\mathcal{AX}_{\texttt{diff}}$ to a theory $\mathcal{AX}_{\texttt{diff}} \cup \mathcal{IDL}$ for which quantifier-free interpolants can also be computed. In particular, we achieve this result via an ad-hoc combination of the procedure for $\mathcal{AX}_{\texttt{diff}}$ outlined in [5] (based on rewriting) and standard methods for solving $\mathcal{IDL}$ constraints (based on a reduction to finding negative cycles in a graph). To the best of our knowledge this is the first successful attempt of combining arrays and (a subset of) arithmetic for obtaining interpolants without quantifiers. The resulting interpolating procedure for $\mathcal{AX}_{\texttt{diff}} \cup \mathcal{IDL}$ may be applied for the verification of programs over arrays, such as sorting algorithms.

The paper is structured as follows. In Section 2, we recall some basic notions of rewriting and first order logic. In Section 3, we introduce the important notion of "modularized constraint". In Section 4, we outline a satisfiability solver for the combination of IDL and a subtheory $\mathcal{BAX}_{\texttt{diff}}$ of $\mathcal{AX}_{\texttt{diff}}$, and we extend it first to produce interpolants (in Section 5), and then to support full $\mathcal{AX}_{\texttt{diff}}$ (in Section 6). We conclude in Section 7. Omitted proofs can be found in the extended version, avaiable at `http://homes.dsi.unimi.it/~ghilardi`.

Related Work. The research on algorithms for computing quantifier-free interpolants for a number of first-order theories has been an active area in the latest years. McMillan proposed in [12] a set of interpolating inference rules to compute interpolants for Linear Rational Arithmetic ($\mathcal{LRA}$), uninterpreted functions ($\mathcal{EUF}$), and their combinations. Alternative approaches, targeted towards efficiency w.r.t. established decision procedures, are based on the lazy framework of [15], and can be found in [9] (for $\mathcal{EUF}$), and in [7, 14] (for $\mathcal{LRA}$ and $\mathcal{LRA} \cup \mathcal{EUF}$). The latter also presents algorithms specific for difference logics ($\mathcal{IDL}$) and unit-two-variables-per-inequality ($\mathcal{UTVPI}$) constraints.

As far as the (classical) theory of arrays ($\mathcal{AX}$) is concerned, it is known [11] that quantifier-free interpolants cannot be computed in general. The same paper suggests a reduction approach to compute interpolants (with quantifiers) for arrays via reduction to uninterpreted functions and linear integer arithmetic ($\mathcal{LIA}$). Unlike [11], our approach is not based on a reduction to other theories. Following the same reduction approach, [3, 4] present an interpolating calculus for computing (in general quantified) interpolants in linear integer arithmetic and some extensions, such as the combination of $\mathcal{LIA}$ with $\mathcal{EUF}$ or $\mathcal{LIA}$ with $\mathcal{AX}$. In contrast, our approach computes quantifier-free interpolants, as we rely on $\mathcal{AX}_{\texttt{diff}}$ instead of $\mathcal{AX}$ as the background theory for modelling arrays. Unlike [4], our approach uses a combination of rewriting and constraint solving as opposed to a sequent calculus, and interpolants are retrieved by means of the application of a set of metarules that record basic transformation steps on the set of constraints. Also, our combination method is *ad hoc*, and it is not based on

a Nelson-Oppen framework as in [15], as the theory of $\mathcal{IDL}$ is non-convex and it thus requires specific treatment. $\mathcal{AX}_{\mathtt{diff}}$ was shown in [5] to have the quantifier-free interpolation property. Here we show that the latter property still holds when combining $\mathcal{AX}_{\mathtt{diff}}$ with $\mathcal{IDL}$.

2 Background and Preliminaries

We assume the usual syntactic (e.g., signature, variable, term, atom, literal, formula, and sentence) and semantic (e.g., structure, truth, satisfiability, and validity) notions of first-order logic. The equality symbol "=" is included in all signatures considered below. For clarity, we shall use "$\equiv$" in the meta-theory to express the syntactic identity between two symbols or two strings of symbols.

Rewriting. We recall some notions and results about term rewriting (see, e.g., [2]) used in the paper. A total ordering $\succ$ on a signature Σ is called a 'precedence' relation. The Lexicographic Path Ordering (LPO) orients equalities by using a given precedence relation; usually, abusing notation, the same symbol is used for the precedence and the associated LPO. Given an equality $s = t$, we write $s \to t$ (called an oriented equality or rewriting rule) when $s \succ t$ for a given precedence $\succ$. Given a set E of oriented equalities, the reduction relation $t \to^* u$ holds when u is obtained by repeatedly rewriting subterms of t by using instances of the rules in E. We say that u is in normal form (w.r.t. a set E of rules) when no rule in E can be applied to u. A set E of rules is *ground irreducible* iff for every ground rule $l \to r$ from E, it is not possible to rewrite neither l nor r by using rules different from $l \to r$ itself. A set E of rules is *convergent* iff every term t has a unique *normal form*, denoted with $\hat{t}$, i.e. $t \to^* \hat{t}$ by using using the rules from E. If the rules in E are all ground and E is ground irreducible, then E is also convergent (because it has no critical pairs, see [2]).

Theories and constraints. A *theory* T is a pair (Σ, Ax_T), where Σ is a signature and Ax_T is a set of Σ-sentences, called the axioms of T (we shall sometimes write directly T for Ax_T). The Σ-structures in which all sentences from Ax_T are true are the *models* of T. A Σ-formula ϕ is *T-satisfiable* if there exists a model $\mathcal{M}$ of T such that ϕ is true in $\mathcal{M}$ under a suitable assignment $\mathtt{a}$ to the free variables of ϕ (in symbols, $(\mathcal{M}, \mathtt{a}) \models \phi$); it is *$T$-valid* (in symbols, $T \vdash \varphi$) if its negation is T-unsatisfiable or, equivalently, iff φ is provable from the axioms of T in a complete calculus for first-order logic. A formula φ_1 *T-entails* a formula φ_2 if $\varphi_1 \to \varphi_2$ is *T-valid*; the notation used for such T-entailment is $\varphi_1 \vdash_T \varphi_2$ or simply $\varphi_1 \vdash \varphi_2$, if T is clear from the context. The *satisfiability modulo the theory T* ($SMT(T)$) *problem* amounts to establishing the T-satisfiability of quantifier-free Σ-formulae.

Let T be a theory in a signature Σ; a *T-constraint* (or, simply, a constraint) A is a set of ground literals in a signature Σ' obtained from Σ by adding a set of free constants. A finite constraint A can be equivalently seen as a single formula, represented by the conjunction of its elements; thus, when we say that a constraint A is *T-satisfiable* (or just "satisfiable" if T is clear from the context), we

mean that the associated formula (also called A) is satisfiable in a Σ'-structure which is a model of T.

Two finite constraints A and B are *logically equivalent* (modulo T) iff $T \vdash A \leftrightarrow B$. The notion of logical equivalence is often too strong when checking T-satisfiability of constraints, as we do in this paper. To overcome this problem, we introduce the notion of $\exists$-equivalence which is weaker than logical equivalence and still implies equisatisfiability of constraints. Let A be a first-order sentence, $A^{\exists}$ is the formula obtained from A by replacing free constants with variables and then existentially quantifying them out.

Definition 1. *Two finite constraints A and B (or, more generally, first order sentences) are* $\exists$-equivalent *(modulo T) iff $T \vdash A^{\exists} \leftrightarrow B^{\exists}$.*

Obviously, the preservation of $\exists$-equivalence is an important requirement for T-satisfiability procedures based on constraint transformations. As an example of such equisatisfiability-preserving transformations based on $\exists$-equivalence, we consider the renaming of terms by constants which will be used in our procedures below. This transformation takes a constraint A and replaces all the occurrences of one of its terms, say t, with a fresh constant a (i.e., a does not occur in A) so to obtain a new constraint A' such that $A' \cup \{a = t\}$ is $\exists$-equivalent to A, where the equality $a = t$ is called the *explicit definition of t*.

Theories of arrays. Let $\mathcal{AX}$ denote the McCarthy theory of arrays with extensionality whose signature contains three sort symbols $\mathtt{ARRAY}, \mathtt{ELEM}, \mathtt{INDEX}$ and two function symbols rd of type $\mathtt{ARRAY} \times \mathtt{INDEX} \longrightarrow \mathtt{ELEM}$ and wr of type $\mathtt{ARRAY} \times \mathtt{INDEX} \times \mathtt{ELEM} \longrightarrow \mathtt{ARRAY}$. The set $\mathcal{AX}$ of axioms contains the following three sentences:

$$\forall y, i, e. \quad rd(wr(y, i, e), i) = e \tag{1}$$

$$\forall y, i, j, e. \ i \neq j \Rightarrow rd(wr(y, i, e), j) = rd(y, j) \tag{2}$$

$$\forall x, y. \ x \neq y \Rightarrow (\exists i. \ rd(x, i) \neq rd(y, i)). \tag{3}$$

It is known [11] that quantifier-free interpolants may not exist for two unsatisfiable quantifier-free formulae in $\mathcal{AX}$. To overcome this problem, in [5], we have introduced the following variant of $\mathcal{AX}$—called the theory of arrays with $\mathtt{diff}$ and denoted with $\mathcal{AX}_{\mathtt{diff}}$—whose signature is that of $\mathcal{AX}$ extended with the function symbol $\mathtt{diff}$ of type $\mathtt{ARRAY} \times \mathtt{ARRAY} \longrightarrow \mathtt{INDEX}$. The set $\mathcal{AX}_{\mathtt{diff}}$ of axioms contains (1), (2), and the following Skolemization of (3):

$$\forall x, y. \ x \neq y \Rightarrow rd(x, \mathtt{diff}(x, y)) \neq rd(y, \mathtt{diff}(x, y)), \tag{4}$$

which constrains the interpretation of $\mathtt{diff}$ to be a (binary) function returning an index at which the input arrays store different values, if such an index exists; otherwise (i.e., when the arrays are identical) $\mathtt{diff}$ returns an arbitrary value. Quantifier-free interpolants can be computed for mutually unsatisfiable quantifier-free formulae of $\mathcal{AX}_{\mathtt{diff}}$ [5].

In this paper, we first consider a sub-theory $\mathcal{BAX}_{\mathtt{diff}}$ of $\mathcal{AX}_{\mathtt{diff}}$ whose signature is that of $\mathcal{AX}_{\mathtt{diff}}$ except for the function symbol wr, which is omitted.

The set of axioms of $\mathcal{BAX}_{\mathtt{diff}}$ is the singleton containing just the Skolemization of extensionality, i.e., (4).

Integer Difference Logic. Following [8], we define $\mathcal{IDL}$ as the mono-sorted theory whose signature contains just one sort symbol, that we call `INDEX` (in preparation to "combine" this theory with $\mathcal{BAX}_{\mathtt{diff}}$ or $\mathcal{AX}_{\mathtt{diff}}$), the constant 0, the binary predicate $\leq$, and two unary function symbols *succ* and *pred*. The axioms of $\mathcal{IDL}$ are all the sentences which are true in the usual structure $\mathbb{Z}$ of the integers when interpreting the constant 0 as the number zero, $\leq$ as the natural ordering, *succ* as the successor ($\lambda x.(x+1)$), and *pred* as the predecessor ($\lambda x.(x-1)$) functions. Ground atoms of $\mathcal{IDL}$ are equivalent to formulæ of the form $i \bowtie S^n(j)$ (where $\bowtie \in \{=, \leq\}$, $S \in \{succ, pred\}$, and i, j are either free constants or 0) and are written as $i - j \bowtie n$ or as $i \bowtie j + n$, for $n \in \mathbb{Z}$. We use also obvious abbreviations like $i \bowtie j$ (for $i - 0 \bowtie j$), $i < j$ (for $i \leq j - 1$), or $i \geq j + n$ (for $j - i \leq -n$), etc. Using these abbreviations, it is easy to see that the theory just defined is usually referred to as "integer difference logic" in the literature.

3 Modularized constraints for the theory $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$

In this paper, we consider the composed theories $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ and $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ (where $\cup$ denotes the union of the signatures and the axioms of the component theories) and design algorithms for the computation of (quantifier-free) interpolants. We do this in two steps. First, we describe an *ad hoc* combination of rewriting (for $\mathcal{BAX}_{\mathtt{diff}}$)—along the lines of [2]—and constraint solving (for $\mathcal{IDL}$) to build a satisfiability procedure and (on top of this) an interpolating algorithm for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$. Then, we show how this can be lifted to compute quantifier-free interpolants also for $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$.

Methodologically, this may appear surprising, but the following observations should clarify our choice. On the one hand, the component theories satisfy the hypotheses of the Nelson-Oppen combination method [13] for satisfiability checking. Furthermore, the Nelson-Oppen method has been extended to combine interpolating procedures in [15] for component theories which are "*equality interpolating*" in order to restrict the formulae to be propagated between interpolating procedures. On the other hand, for simplicity, the definition of equality interpolating theory in [15] applies only to convex theories; unfortunately, $\mathcal{IDL}$ is not convex and we were not able to extend the notion of equality interpolating to the non-convex case in a form that applies to our case. Our experience suggests that finding the right generalization of this notion is far from being trivial and explains why we preferred to design the *ad hoc* method in this paper. More precisely, after a pre-processing phase, we separate constraints in two parts: one pertaining to $\mathcal{BAX}_{\mathtt{diff}}$ and one to $\mathcal{IDL}$. Literals in $\mathcal{BAX}_{\mathtt{diff}}$ are transformed by ground rewriting (along the lines of [5]) while for those in $\mathcal{IDL}$, we adapt available constraint solving techniques for integer difference logic. The goal of these transformations is to derive a (so-called) *modularized* constraint whose satisfiability is trivial to establish.

Let $a, b, \ldots$ denote free constants of sort $\texttt{ARRAY}$, $i, j, \ldots$ free constants of sort $\texttt{INDEX}$, $d, e, \ldots$ free constants of sort $\texttt{ELEM}$, and $\alpha, \beta, \ldots$ free constants of any sort. A *(ground) flat* literal is a literal of the form $i \bowtie j + n, rd(a, i) = e, \texttt{diff}(a, b) = i, \alpha = \beta, \alpha \neq \beta$. By replacing literals of the form $i \not\leq j$ with $j \leq i - 1$ and renaming terms with constants as explained in Section 2, given a constraint A it is always possible to produce an $\exists$-equivalent (*flat*) constraint A' such that A' contains only flat literals. We first analyze in detail flat constraints; we introduce some notions, aiming at defining, so called, "modularized" constraints for which satisfiability can be easily assessed both in isolation and combination.

Separation. We split a flat constraint A in two: the *index* part A_I and the *main* part A_M, where A_I contains the literals of the form $i = j + n, i \leq j + n, i \neq j, \texttt{diff}(a, b) = i$ and A_M contains the remaining literals, i.e., those of the forms $a = b, a \neq b, rd(a, i) = e, e = d, e \neq d$.

Rewriting for $\mathcal{BAX}_{\texttt{diff}}$. We fix the precedence $\succ$ to be such that $\leq \succ a \succ rd \succ \texttt{diff} \succ i \succ succ \succ pred \succ 0 \succ e$, for every a, i, e of the corresponding sorts. The LPO extension of $\succ$ allows us to orient all the equalities in the main part A_M of a constraint (in particular, we have that $rd(a, i) = e$ is oriented as $rd(a, i) \to e$, and $\alpha = \beta$ is oriented as $\alpha \to \beta$ when $\alpha \succ \beta$).

Constraint solving for $\mathcal{IDL}$. Equalities in the index part A_I are classified as follows: a $\texttt{diff}$-*explicit definition* is an equality having the form $\texttt{diff}(a, b) = i$ and an $\mathcal{IDL}$-*explicit definition* is an equality of the form $i = j + n$ (with $i \not\equiv j, i \not\equiv 0$). Each equality in A_I can be rewritten as an $\mathcal{IDL}$-explicit definition, unless it is a $\texttt{diff}$-explicit definition, or a tautology (such as $i = i + 0$), or it is unsatisfiable (as $i = i + 4$). In an $\mathcal{IDL}$-explicit definition $i = j + n$, we say that "i is explicitly defined by $j + n$" (notice that j can be 0). As it is customary in solvers for difference logic (see, e.g. [1]), we associate the integer difference logic literals of the form $j \leq i + n$ with a weighted directed graph. More precisely, let $G(V, E)$ be a (finite, integer-weighted, directed) graph: the notation $i \xrightarrow{n} j$ means that there is an edge from i to j with weight n. We can associate the tuple $\langle G_A(V_A, E_A), \mathcal{D}_A, d_A, n_A \rangle$ to the index part A_I, where $\mathcal{D}_A$ is a set of $\texttt{diff}$-explicit definitions, d_A is a set of $\mathcal{IDL}$-explicit definitions, n_A is a set of negated equalities, and $G_A(V_A, E_A)$ has an edge $i \xrightarrow{n} j \in E_A$, for each $j - i \leq n \in A_I$.

Constraints in $\mathcal{BAX}_{\texttt{diff}} \cup \mathcal{IDL}$. We write a constraint A as follows:

$$A = \langle G_A, \mathcal{D}_A, d_A, n_A, A_M \rangle \ . \tag{5}$$

By abusing notation, we confuse the graph G_A with the corresponding set of inequalities and leave implicit both the set V_A of vertices and E_A of edges.

Definition 2 (Modularized constraint). *A* modularized *constraint is a flat constraint of the form* (5) *such that the following conditions are satisfied:*

(i) d_A *is* appropriate *in the sense that: (a) each free constant has at most one definition; (b) different constants have different definitions; (c) no constant is defined as another constant (i.e. as* $j+0$*); (d) 0 is not defined; (e) defined constants do not occur as vertices in* G_A*;*
(ii) *the graph* G_A *is* acyclic*;*
(iii) *the rewrite system formed by the equalities in* A_M *is convergent and ground irreducible (below, the normal form of a term* t *w.r.t.* A_M *is denoted by* $\hat{t}$*); the array inequalities in* A_M *do not contain* $a \neq b$*;*
(iv) $\{\mathtt{diff}(a,b) = i, \mathtt{diff}(a',b') = i'\} \subseteq \mathcal{D}_A$*,* $\hat{a} \equiv \widehat{a'}$*, and* $\hat{b} \equiv \widehat{b'}$ *imply* $i \equiv i'$*;*
(v) $\mathtt{diff}(a,b) = i \in \mathcal{D}_A$ *and* $\widehat{rd(a,i)} \equiv \widehat{rd(b,i)}$ *imply* $\hat{a} \equiv \hat{b}$*.*

Notice that no index equality $i = j$ may occur in a modularized constraint because of condition (i)(c) above. Also notice that condition (iii) is much stronger than what is usually required for satisfiability. The reason is that we want the satisfiability of modularized constraints to be invariant under addition of (implicit) inequalities.

Proposition 1. *Suppose that* A *is modularized and that there is no element inequality* $e \neq d$ *in* A_M *such that* $\hat{e} \equiv \hat{d}$*. Then* $A \cup \{\alpha \neq \beta\}_{\alpha,\beta}$ *(varying* α, β *among the different pairs of constants in normal form of the same sort occurring in* A*) is* $(\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL})$*-satisfiable.*

Finally, conditions (iv) and (v) of Definition 2 deal with `diff`-explicit definitions: the former requires `diff` to be "well-defined" and the latter is a "conditional" reformulation of the contrapositive of axiom (4).

Combining modularized constraints. Let A, B be two constraints in the signatures Σ^A, Σ^B obtained from the signature Σ by adding some free constants and let $\Sigma^C := \Sigma^A \cap \Sigma^B$. Given a term, a literal, or a formula φ we call it:

- AB*-common* iff it is defined over Σ^C;
- A*-local* (resp. B*-local*) if it is defined over Σ^A (resp. Σ^B);
- A*-strict* (resp. B*-strict*) iff it is A-local (resp. B-local) but not AB-common;
- AB*-mixed* if it contains symbols in both $(\Sigma^A \setminus \Sigma^C)$ and $(\Sigma^B \setminus \Sigma^C)$;
- AB*-pure* if it does not contain symbols in both $(\Sigma^A \setminus \Sigma^C)$ and $(\Sigma^B \setminus \Sigma^C)$.

(Sometimes in the literature about interpolation, "A-local" and "B-local" are used to denote what we call here "A-strict" and "B-strict"). As we will see below, the following modularity result is crucial for interpolation.

Proposition 2. *Let* Σ *be the signature of* $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$*,* $A = \langle G_A, \mathcal{D}_A, d_A, n_A, A_M \rangle$*, and* $B = \langle G_B, \mathcal{D}_B, d_B, n_B, B_M \rangle$ *be modularized constraints in the expanded signatures* Σ^A, Σ^B*. We have that* $A \cup B$ *is modularized in case the four conditions below are all satisfied:*

(O) *the restriction of* A *and* B *to the common subsignature* $\Sigma^C := \Sigma^A \cap \Sigma^B$ *coincide;*
(I) *for each* $\mathcal{IDL}$*-explicit definition* $i = j + n \in d_A \cup d_B$*, if* $i \in \Sigma^C$ *then* $j \in \Sigma^C$*;*

(II) *given $c, c' \in \Sigma_C$, there is a path in G_A leading from c to c' iff there is a path in G_B leading from c to c';*
(III) *for each equation (or rule) of the form $\alpha = t$ in $A \cup B$, if the term t is AB-common, then so is the constant α.*

4 A satisfiability solver for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$

Given a (finite) constraint A in $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$, we present a sequence of transformations for deriving a set $\{A_i \mid 1 \leq i \leq n\}$ of modularized constraints such that $\bigvee_{1 \leq i \leq n} A_i$ is $\exists$-equivalent to A. Since the satisfiability of each A_i is easy to check, we can thus establish the satisfiability of the original constraint A. The transformations are closely related to those in [5] for the theory $\mathcal{AX}_{\mathtt{diff}}$. This approach has two advantages. First, it allows us to use the same method of [5] to lift the satisfiability solver to an interpolating solver. Second, as we will see in Section 6, it is easy to lift the interpolating solver for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ to the theory $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$.

The satisfiability solver for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ consists of the three groups of transformations below applied to the input constraint A.

4.1 Preprocessing

This group consists of the following steps to be executed sequentially.

Step 1 Flatten A, by replacing sub-terms with fresh constants and adding the related defining equalities; replace also literals of the form $i \not\leq j$ with $j \leq i - 1$.

Step 2 Replace array inequalities $a \neq b$ by the following literals

$$\mathtt{diff}(a, b) = i, \quad rd(b, i) = e, \quad rd(a, i) = d, \quad d \neq e,$$

where i, e, d are fresh constants.

Step 3 Guess a total ordering on the index constants occurring in the constraint obtained from the application of the previous two steps. That is, for each pair (i, j) of indexes, add either $i = j$, $i \leq j - 1$, or $j \leq i - 1$. Remove the positive literals $i = j$ by replacing all occurrences of i with j if $i \succ j$ (according to the symbol precedence); otherwise, replace each occurrence of j with i. Now, if an unsatisfiable literal $i \neq i$ is derived, then try another guess. If all guesses produce an unsatisfiable literal, then return `unsat`isfiability; otherwise, each negative literal $i \neq j$ (for each $i \not\equiv j$) is now redundant and can be removed.

Step 4 For each pair (a, i) of constants such that $rd(a, i) = e$ does not occur in the current constraint, add the literal $rd(a, i) = e$ with e fresh constant.

It is easy to see that these four steps terminate and that we obtain a finite set $\{A_i \mid 1 \leq i \leq n\}$ of flat constraints, whose disjunction is $\exists$-equivalent to the

original constraint A. If there exists $i \in \{1, ..., n\}$ such that the exhaustive applications of the transformations in the next group (Completion) does not exit with a `failure`, then return `sat`isfiable; otherwise (i.e. for each $i \in \{1, ..., n\}$, the transformations in the next group halt with a `failure`), report `unsat`isfiable.

4.2 Completion

Let $\langle G_A, \mathcal{D}_A, d_A, n_A, A_M \rangle$ be one of the n flat constraints obtained from Preprocessing. We exhaustively apply to this constraint the following group of rules, which are organized in three sub-groups to clarify their purposes. All the transformations below can be interleaved arbitrarily; they are all deterministic with the exception of (G2) which might introduce further case-splits.

(I) Graph Completion. The transformations in this sub-group aim at satisfying conditions (i) and (ii) of Definition 2. We view G_A in a constraint as both a graph and a set of inequalities (recall that an arc $i \xrightarrow{n} j$ represents the inequality $j - i \leq n$): when some inequalities in the constraint are modified, the graph is updated accordingly.

(G1) Suppose we have an $\mathcal{IDL}$-explicit definition $i = j + n$. If $i \equiv j$, then the literal is either trivially true and can be removed, or false and `failure` can be reported. Otherwise, i.e. if $i \succ j$ (when $j \succ i$, we can rewrite it to $j = i - n$), keep the equality $i = j + n$ and also the equalities of the kind $\texttt{diff}(a, b) = i$, but replace every other occurrence of i in the index part of the current constraint by $j + n$ (it is easy to see that the constraint remains flat, after normalization of ground atoms, if needed).

(G2) Suppose we have a cycle $i_1 \xrightarrow{n_1} i_2 \xrightarrow{n_2} i_3 \cdots i_k \xrightarrow{n_k} i_1$ in G_A. If $n_1 + \cdots + n_k < 0$, then report `failure`. If $k = 1$, then since $n_1 \geq 0$, the arc represents a tautology and can be removed. In case $k > 1$, we can assume that i_1 is the $\succ$-biggest node in the cycle (if this is not the case, a permutation of the cycle is sufficient to satisfy this assumption) and that i_1 does not have an $\mathcal{IDL}$-explicit definition (otherwise instruction (G2) is not applied, (G1) should be applied instead). Let $m := n_2 + \cdots + n_k$; the cycle entails that i_1 lies in the integer interval $[i_2 - n_1, i_2 + m]$ (i.e. it entails $i_1 = i_2 - n_1 \vee i_1 = i_2 - n_1 + 1 \vee \cdots \vee i_1 = i_2 + m$) and we can add to the current constraint an $\mathcal{IDL}$-explicit definition for i_1 via a disjunctive guessing.

(II) Knuth-Bendix Completion. The transformations in this sub-group aim at satisfying condition (iii) of Definition 2 by using a Knuth-Bendix completion process (see, e.g., [2]). In particular, (K1)-(K3) remove critical pairs.

(K1) $\boxed{d \ {}_*\!\leftarrow rd(b, i) \leftarrow rd(a, i) \rightarrow e' \rightarrow_* e}$

Remove the parent rule $rd(a, i) \rightarrow e'$ and keep the other parent rule $a \rightarrow b$. If $d > e$ (resp. $e > d$), then add the rule $d \rightarrow e$ (resp. $e \rightarrow d$); otherwise (i.e. when $d \equiv e$), do nothing. (Notice that terms of the form $rd(b, i)$ are always reducible to an element constant because of Step 4 in the pre-processing phase.)

(K2) $\boxed{e \ {}_*\!\leftarrow e' \leftarrow rd(a,i) \rightarrow d' \rightarrow_* d}$

If $e \not\equiv d$, then orient the critical pair, add it as a new rule, and remove one of the parent rules.

(K3) $\boxed{\alpha \ {}_*\!\leftarrow \alpha' \leftarrow \beta \rightarrow \beta'_1 \rightarrow_* \beta_1}$

If $\beta \not\equiv \beta_1$, then orient the critical pair, add it as a new rule, and remove one of the parent rules; here $\alpha, \alpha', \beta, \beta_1, \beta'_1$ are all either of sort `ARRAY` or of sort `ELEM`.

(K4) If the right-hand side of a current ground rewrite rule can be reduced, then reduce it as much as possible, remove the old rule, and replace it with the newly obtained reduced rule.

(K5) If there exists a negative literal $e \neq d \in A_M$ such that $e \rightarrow_* e' \ {}_*\!\leftarrow d$, then report `failure`.

(III) Handling `diff`***.*** The transformations in this group take care of condition (iv)-(v) of Definition 2 (we write $t \downarrow t'$ to mean that $t \rightarrow_* u \ {}_*\!\leftarrow t'$ for some u).

(S) If $\mathtt{diff}(a,b) = i \in A_I$, $rd(a,i) \downarrow rd(b,i)$ and $a \succ b$, then add the rule $a \rightarrow b$ and replace $\mathtt{diff}(a,b) = i$ by $\mathtt{diff}(b,b) = i$ (this is needed for termination, it prevents the rule from being indefinitely applied).

(U) If $\{\mathtt{diff}(a,b) = i, \mathtt{diff}(a',b') = i'\} \subseteq A_I$, $a \downarrow a'$ and $b \downarrow b'$ for $i \not\equiv i'$, then report `failure` and backtrack to Step 3 of the pre-processing phase.

It can be proved that the algorithm using the groups of transformations described above terminates and computes modularized constraints. We are now in the position to derive the main result of this section:

Theorem 1. *Every constraint in $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ is $\exists$-equivalent to a disjunction of modularized constraints. The algorithm described above decides ($\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$)-satisfiability.*

5 An interpolating solver for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$

Our design of the interpolating solver (as in [5]) is based on an abstract framework, in which we focus on the *basic operations* necessary to derive an interpolating refutation, independently of the underlying satisfiability procedure.

5.1 Interpolating Metarules

Let A, B be constraints in signatures Σ^A, Σ^B expanded with free constants and $\Sigma^C := \Sigma^A \cap \Sigma^B$. Recall the definitions of AB-common, A-local, B-local, A-strict, B-strict, AB-mixed, AB-pure terms, literals and formulae given in Section 3. The goal is to compute a ground AB-common sentence ϕ such that $A \vdash_{\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}} \phi$ and $\phi \wedge B$ is ($\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$)-unsatisfiable, whenever $A \wedge B$ is ($\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$)-unsatisfiable.

The basic operations needed to re-design the solver of Section 4 in order to add the capability of computing interpolants are called *metarules* and are shown

in Table 1 (in this section, we use $\phi \vdash \psi$ for $\phi \vdash_{\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}} \psi$). The metarules are the same as those introduced in [5] to which the reader is pointed for more details.

Close1	Close2	Propagate1	Propagate2
$\dfrac{}{A \mid B}$	$\dfrac{}{A \mid B}$	$\dfrac{A \mid B \cup \{\psi\}}{A \mid B}$	$\dfrac{A \cup \{\psi\} \mid B}{A \mid B}$
Prov.: A is unsat. *Instr.*: $\phi' \equiv \bot$.	*Prov.*: B is unsat. *Instr.*: $\phi' \equiv \top$.	*Prov.*: $A \vdash \psi$ and ψ is AB-common. *Instr.*: $\phi' \equiv \phi \wedge \psi$.	*Prov.*: $B \vdash \psi$ and ψ is AB-common. *Instr.*: $\phi' \equiv \psi \rightarrow \phi$.

Define0	Define1	Define2
$\dfrac{A \cup \{a = t\} \mid B \cup \{a = t\}}{A \mid B}$	$\dfrac{A \cup \{a = t\} \mid B}{A \mid B}$	$\dfrac{A \mid B \cup \{a = t\}}{A \mid B}$
Prov.: t is AB-common, a fresh. *Instr.*: $\phi' \equiv \phi(t/a)$.	*Prov.*: t is A-local and a is fresh. *Instr.*: $\phi' \equiv \phi$.	*Prov.*: t is B-local and a is fresh. *Instr.*: $\phi' \equiv \phi$.

Disjunction1	Disjunction2
$\dfrac{\cdots \quad A \cup \{\psi_k\} \mid B \quad \cdots}{A \mid B}$	$\dfrac{\cdots \quad A \mid B \cup \{\psi_k\} \quad \cdots}{A \mid B}$
Prov.: $\bigvee_{k=1}^{n} \psi_k$ is A-local and $A \vdash \bigvee_{k=1}^{n} \psi_k$. *Instr.*: $\phi' \equiv \bigvee_{k=1}^{n} \phi_k$.	*Prov.*: $\bigvee_{k=1}^{n} \psi_k$ is B-local and $B \vdash \bigvee_{k=1}^{n} \psi_k$. *Instr.*: $\phi' \equiv \bigwedge_{k=1}^{n} \phi_k$.

Redplus1	Redplus2	Redminus1	Redminus2
$\dfrac{A \cup \{\psi\} \mid B}{A \mid B}$	$\dfrac{A \mid B \cup \{\psi\}}{A \mid B}$	$\dfrac{A \mid B}{A \cup \{\psi\} \mid B}$	$\dfrac{A \mid B}{A \mid B \cup \{\psi\}}$
Prov.: $A \vdash \psi$ and ψ is A-local. *Instr.*: $\phi' \equiv \phi$.	*Prov.*: $B \vdash \psi$ and ψ is B-local. *Instr.*: $\phi' \equiv \phi$.	*Prov.*: $A \vdash \psi$ and ψ is A-local. *Instr.*: $\phi' \equiv \phi$.	*Prov.*: $B \vdash \psi$ and ψ is B-local. *Instr.*: $\phi' \equiv \phi$.

ConstElim1	ConstElim2	ConstElim0
$\dfrac{A \mid B}{A \cup \{a = t\} \mid B}$	$\dfrac{A \mid B}{A \mid B \cup \{b = t\}}$	$\dfrac{A \mid B}{A \cup \{c = t\} \mid B \cup \{c = t\}}$
Prov.: a is A-strict and does not occur in A, t. *Instr.*: $\phi' \equiv \phi$.	*Prov.*: b is B-strict and does not occur in B, t. *Instr.*: $\phi' \equiv \phi$.	*Prov.*: c, t are AB-common, c does not occur in A, B, t. *Instr.*: $\phi' \equiv \phi$.

Table 1. Interpolating Metarules: each rule has a proviso *Prov.* and an instruction *Instr.* for recursively computing the new interpolant ϕ' from the old one(s) $\phi, \phi_1, \ldots, \phi_k$. Metarules are applied *bottom-up* and interpolants are computed *top-down*.

The correctness of the procedure explained in Subsection 5.2 relies on Proposition 3 below. Before stating the proposition, we need to introduce the following formal notion. An *interpolating metarules refutation* for A, B is a labeled tree having the following properties: (i) nodes are labeled by pairs of finite sets of constraints; (ii) the root is labeled by A, B; (iii) the leaves are labeled by a pair $\tilde{A}, \tilde{B}$ such that $\bot \in \tilde{A} \cup \tilde{B}$; (iv) each non-leaf node is the conclusion of a rule from Table 1 and its successors are the premises of that rule.

Proposition 3 ([5]). *If there exists an interpolating metarules refutation for A, B then there is a quantifier-free interpolant for A, B (i.e. there exists a quantifier-free AB-common sentence ϕ such that $A \vdash \phi$ and $B \wedge \phi \vdash \bot$). The interpolant ϕ is recursively computed by applying the relevant interpolating instructions from Table 1.*

Metarules are useful to design an algorithm manipulating pairs of constraints based on transformation instructions. Each of the transformation instructions is derived from the satisfiability solver of Section 4 and is *justified* by a metarule (or by a sequence of metarules): in this way, if our instructions form a complete and terminating algorithm, we can use Proposition 3 to get the desired interpolants. The main advantage of this approach is that we just need to take care of the completeness and termination of the algorithm, while ignoring interpolants. Here "completeness" means that our transformations should be able to bring a pair (A, B) of constraints into an $\exists$-equivalent set of pairs of constraints (A_i, B_i) that either match the requirements of Proposition 2 or are trivially unsatisfiable (i.e. $\bot \in A_i \cup B_i$). By Theorem 1, the latter happens iff the original pair (A, B) is $(\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL})$-unsatisfiable or, equivalently, we get an interpolating metarules refutation.

5.2 The Interpolating Solver

The key idea for lifting the satisfiability solver of Section 4 to an interpolating solver is that of invoking it separately on A and B, and propagating equalities involving AB-common terms. We shall use *a precedence in which AB-common constants are smaller than A-strict or B-strict constants of the same sort*. Unfortunately, this is not sufficient to prevent the instances of the satisfiability solver from generating literals and rules violating one or more of the hypotheses of Proposition 2. This is the reason for introducing further correcting instructions (γ)-(δ) below. The interpolating solver consists of two groups of instructions, detailed below and called pre-processing and completion, derived from those in Sections 4.1 and 4.2. In the following, the A-component and the B-component of the constraints under consideration will be called A and B.

Pre-processing. This group of transformations contains those in Section 4.1. They are performed on both A and B. To justify these transformations, we need metarules (Define0,1,2), (Redplus1,2), (Redminus1,2), (Disjunction1,2), (ConstElim0,1,2), and (Propagate1,2) in Table 1. The last two are required because when i and j are AB-common, the case-splitting on $i = j$, $i < j$, or $j < i$ of

Step 3 can be done—say—in A and then propagated to B. After the applications of these transformations, the following invariants (to be maintained also by the next group of transformations) hold:

(i1) A (resp. B) entails (modulo $\mathcal{BAX}_{\texttt{diff}} \cup \mathcal{IDL}$) either $i < j$ or $j < i$, for each A-local (resp. B-local) constants i, j of sort `INDEX` in A (resp. B);
(i2) if the constants a, i occur in A (resp. in B), then $rd(a, i)$ reduces to an A-local (resp. B-local) constant of sort `ELEM`.

Completion. This group of transformations are executed non-deterministically until no more rules can be applied.

(α) Apply any instruction of Section 4.2 to A or B.
(β) If there is an AB-common literal that belongs to A but not to B (or vice versa), copy it to B (resp. A).
(γ) "Repair" (see below for a precise description) those literals violating conditions (I) or (III) of Proposition 2, called *undesired literals* below.
(δ) If G_A (or G_B) contains a path $i_1 \xrightarrow{n_1} i_2 \xrightarrow{n_2} i_3 \cdots i_k \xrightarrow{n_k} i_{k+1}$ between AB-common constants i_1 and i_{k+1} and there is no path from i_1 to i_{k+1} in $G_A \cap G_B$, then add the inequality $i_{k+1} - i_1 \leq n_1 + \cdots + n_k$ to both A and B (so that an arc from i_1 to i_{k+1} will be created in $G_A \cap G_B$).

Instructions in (α) deleting an AB-common literal should be performed *simultaneously* in A and B. It can be easily checked (the check is done within the proof of Theorem 2) that this is always possible by inspecting the transformations in Section 4.2. An easy way to guarantee this is to give higher priority to the rules in (β) and (γ).

Preliminary to describing how to "repair" literals—i.e. the instructions in (γ)—we need to introduce a technique that we call *Term Sharing*. Suppose that A contains a literal $\alpha = t$ where the term t is AB-common but the free constant α is only A-local. It is possible to "make α AB-common" as follows. First, introduce a fresh AB-common constant α' with the explicit definition $\alpha' = t$ (to be inserted both in A and in B, as justified by metarule (Define0)). Then, replace the literal $\alpha = t$ with $\alpha = \alpha'$ and α with α' everywhere else in A. Finally, delete $\alpha = \alpha'$. The result is a pair (A, B) of constraints which is almost identical to the original pair except for the fact that α has been renamed to an AB-common constant α'. These transformations can be justified by metarules (Define0), (Redplus1), (Redminus1), (ConstElim1). This concludes the description of Term Sharing.

We are now in the position to explain the instructions in (γ): notice that literals violating conditions (I) or (III) of Proposition 2 are all of the form $\alpha = t$, where t is AB-common and α is, say, just A-local (this applies also to the literals $i = j + n$ violating (I), because they can be rewritten as $j = i - n$). Clearly, Term Sharing can replace them by literals of the form $\alpha' = t$, where α' is AB-common too. There is however a subtlety to take care of in case α is of sort `INDEX`: since α' is AB-common whereas α is only A-local, we might need to perform some guessing to maintain invariant (i1). In other words, we need to repeat Step 3

from Section 4.1 until invariant (i1) is restored (α' must be compared with the other B-local constants of sort `INDEX`).

By exhaustively applying the transformations in the two groups above (namely, Pre-processing and Completion) on a pair (A, B) of constraints, we can produce a tree whose nodes are labelled by pairs of constraints and such that the successor nodes are labelled by pairs of constraints that are obtained by applying an instruction. We call such a tree an *interpolating tree* for (A, B). The key observation is that interpolating trees are interpolating metarules refutation trees when input pairs of constraints are mutually unsatisfiable.

Theorem 2. *Any interpolating tree for (A, B) is finite and it is an interpolating metarules refutation iff $A \wedge B$ is $(\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL})$-unsatisfiable.*

By using this theorem and recalling Proposition 3, a (quantifier-free) interpolant can be recursively computed by using the metarules of Table 1. In other words, we have designed a quantifier-free interpolating solver for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$.

Theorem 3. *$\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ admits quantifier-free interpolants (i.e., for every quantifier free formulae ϕ, ψ such that $\psi \wedge \phi$ is $(\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL})$-unsatisfiable, there exists a quantifier free formula θ such that: (i) $\psi \vdash_{\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}} \theta$; (ii) $\theta \wedge \phi$ is not $(\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL})$-satisfiable; and (iii) only the variables occurring both in ψ and ϕ occur also in θ).*

6 An interpolating solver for $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$

We now sketch how to lift the interpolating solver for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ to one for $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ by combining the results from Section 5 with those of [5]. Since $\mathcal{BAX}_{\mathtt{diff}}$ is a sub-theory of $\mathcal{AX}_{\mathtt{diff}}$, it is straightforward to reuse the rewriting approach of [5] to extend the solver outlined in the previous section. Because of lack of space, we only describe the key ideas and we point the reader to [5] for details about the solver for $\mathcal{AX}_{\mathtt{diff}}$. The difference between $\mathcal{BAX}_{\mathtt{diff}}$ and $\mathcal{AX}_{\mathtt{diff}}$ is in the presence of the function symbol wr and the axioms (1) and (2). We explain how the rewriting techniques of [5], used to cope with terms consisting of nested wr's, can be seen as an extension of those used in this paper. We recall the following notation from [5]: $wr(a, I, E)$ abbreviates the term $wr(wr(\cdots wr(a, i_1, e_1) \cdots), i_n, e_n)$, i.e., a nested write on the array variable a where indexes and elements are represented by the free constants lists $I \equiv i_1, \ldots, i_n$ and $E \equiv e_1, \ldots, e_n$, respectively.

We extend our precedence in such a way that $a \succ wr \succ rd \succ \mathtt{diff} \succ i$ holds, for all constants a, i. This condition (satisfied by the precedence adopted in [5]) implies that an equality $a = wr(b, I, E)$ can be turned to a rewrite rule of the form $a \rightarrow wr(b, I, E)$ when $a \succ b$. As explained in [5], this is crucial to design an extended version of the Knuth-Bendix completion (see Section 4.2 of this paper), which allows for computing modularized constraints in $\mathcal{AX}_{\mathtt{diff}}$. Intuitively, the Knuth-Bendix completion is added transformations for eliminating "badly orientable" equalities (i.e. equalities of the form $b = wr(a, I, E)$ with $a \succ b$) that

may arise. Such transformations solve the equality $b = wr(a, I, E)$ for a, thereby deriving a rewriting rule $a \to wr(b, I, E')$ for suitable E'.

Recall from [5] that an $\mathcal{AX}_{\mathtt{diff}}$ flat constraint contains only literals of the forms $\alpha = \beta, \alpha \neq \beta, rd(a, i) = e, \mathtt{diff}(a, b) = i$, and $b = wr(a, I, E)$. An $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ constraint is *flat* iff its restrictions to the signatures of $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ and $\mathcal{AX}_{\mathtt{diff}}$ are flat.

Definition 3. *An $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ flat constraint is* modularized *iff (a) its restriction to the signature of $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ is modularized (according to Definition 2) and (b) its restriction to the signature of $\mathcal{AX}_{\mathtt{diff}}$ is modularized according to Definition 3.1 of [5].*

The most important additional requirements of Definition 3.1 in [5] induce irredundant normal forms for terms built out of rd's and wr's by means of a set of non-ground rewrite rules corresponding to the axioms (1) and (2).

By "merging" the proof of Proposition 1 and that of the corresponding result in [5], we can show that the satisfiability of modularized constraint is invariant under addition of (implicit) inequalities, i.e. that *Proposition 1 holds also for $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$.* By "merging" the proofs of Proposition 2 and Proposition 3.3 in [5], we can "combine" modularized $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ constraints.

Proposition 4. *Let A and B be $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ modularized constraints in expanded signatures Σ^A, Σ^B (here Σ is the signature of $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$). We have that $A \cup B$ is modularized in case the conditions (O)-(III) of Proposition 2 and the conditions (O)-(III) of Proposition 3.3 from [5] are both satisfied.*

At this point, we have all the ingredients to design first a satisfiability solver for $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ and then to extend it to an interpolating solver. The first step consists of the merging of the transformations in the groups pre-processing and completion of Section 4 with those in Section 4 of [5]. It is then possible to prove that the resulting algorithm checks for $(\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL})$-satisfiability by "merging" the proofs of Theorem 1 and Theorem 4.1 of [5].

The second step amounts to integrate the instructions of the interpolating solver from Section 5.2 with those of the interpolating solver from Section 5.2 in [5] (notice that the interpolating metarules used in this paper are identical to those in [5]). The most important addition is the repairing of undesired literals of the form $c \to wr(c', I, E)$ whose left-hand side is AB-common but whose right-hand side is, say, only A-local: repairing requires a careful splitting of I and E into sub-lists, additional guessings, and manipulations of nested wr's similar to those for eliminating badly orientable equations (see Section 5.2 in [5] for details). It is then possible to build interpolating trees (defined in a similar way as those in Section 5.2).

Theorem 4. *The theory $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ admits quantifier-free interpolants (in the sense of Theorem 3 where $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ is replaced with $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$).*

The proof of the above theorem is obtained by a straightforward merging of the proofs of the corresponding results for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ and $\mathcal{AX}_{\mathtt{diff}}$ (the complexity measures from the termination arguments are essentially the same).

7 Conclusions and future work

In this work we have shown how to derive an interpolating (satisfiability) solver for the theory of $\mathcal{AX}_{\texttt{diff}} \cup \mathcal{IDL}$. Most importantly, the produced interpolants are quantifier-free: we are not aware of any other approach that can derive interpolants for arrays and arithmetic without introducing quantifiers. Thus our work can find suitable applications in existing verification techniques based on abstraction-refinement loops with the help of interpolants.

In order to achieve our result, we have combined rewriting techniques for two variants of the theory of arrays, $\mathcal{BAX}_{\texttt{diff}}$ and $\mathcal{AX}_{\texttt{diff}}$, with a constraint solver for $\mathcal{IDL}$ based on a reduction to graph algorithms. Interpolants may be computed with the help of interpolating metarules to be applied in reverse order w.r.t. the algorithmic transformations steps.

We plan to implement our approach in the SMT solver OpenSMT [6] in order to carry out an extensive experimental evaluation.

References

1. B. Cherkassky and A. Goldberg. Negative-cycle Detection Algorithms. In *Algorithms ESA '96*, pages 349–363. 1996.
2. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, Cambridge, 1998.
3. A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic . In *IJCAR*, 2010.
4. A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. Beyond Quantifier-Free Interpolation in Extensions of Presburger Arithmetic. In *VMCAI*, 2012. to appear.
5. R. Bruttomesso, S. Ghilardi, and S. Ranise. Rewriting-based Quantifier-free Interpolation for a Theory of Arrays. In *RTA*, 2011.
6. R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The OpenSMT Solver. In *TACAS*, pages 150–153, 2010.
7. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Interpolation Generation in Satisfiability Modulo Theories. *ACM Trans. Comput. Logic*, 12:1–54, 2010.
8. Herbert B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York-London, 1972.
9. A. Fuchs, A. Goel, J. Grundy, S. Krstić, and C. Tinelli. Ground Interpolation for the Theory of Equality. In *TACAS*, pages 413–427, 2009.
10. T. Henzinger and K. L. McMillan R. Jhala, R. Majumdar. Abstractions from Proofs. In *POPL*, 2004.
11. D. Kapur, R. Majumdar, and C. Zarba. Interpolation for Data Structures. In *SIGSOFT'06/FSE-14*, pages 105–116, 2006.
12. K. L. McMillan. An Interpolating Theorem Prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.
13. G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.
14. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. In *VMCAI*, 2007.
15. G. Yorsh and M. Musuvathi. A Combination Method for Generating Interpolants. In *CADE*, pages 353–368, 2005.