AXDInterpolator: A Tool for Computing Interpolants for Arrays with MaxDiff

Jose Abel Castellanos Joo[1], Silvio Ghilardi[2], Alessandro Gianola[3] and Deepak Kapur[1]

[1]*Department of Computer Science, University of New Mexico, Albuquerque, USA*
[2]*Dipartimento di Matematica, Università degli Studi di Milano, Milan, Italy*
[3]*Faculty of Computer Science, Free University of Bozen-Bolzano, Bolzano, Italy*

Abstract

Several approaches toward quantifier-free interpolation algorithms of theories involving arrays have been proposed by extending the language using a binary function skolemizing the extensionality principle. In FoSSaCS 2021, the last three authors studied the enrichment of the McCarthy's theory of extensional arrays with a maxdiff operation. This paper discusses the implementation of the interpolation algorithm proposed in FoSSaCS 2021 using the `Z3` API. The implementation allows the user to choose `iZ3`, `Mathsat`, or `SMTInterpol` as interpolation engines. The tool returns a formula in SMTLIB2 format, which allows compatibility with model checkers and invariant generators using such a format. We compare our algorithm with state-of-the-art interpolation engines. Our experiments using unsatisfiable formulæ extracted with the model checker UAutomizer show the feasibility of our tool. For that purpose, we used C programs from the ReachSafety-Arrays and MemSafety-Arrays tracks of SV-COMP.

Keywords

Interpolation, Arrays, MaxDiff, SMT

1. Introduction

Since McMillan's seminal papers [1, 2], interpolation has been successfully applied in software model checking. Even if the well-known Craig's theorem [3] guarantees the existence of interpolants in first order logic, it does not give any information on the shape the interpolant can have when a specific theory is involved. Nevertheless, this is crucial for the applications: when we extract interpolants, we are typically handling a theory which might be undecidable, but whose quantifier-free fragment is decidable for satisfiability (usually within a somewhat 'reasonable' computational complexity). Thus, it is desirable (although not always possible) that the interpolant is quantifier-free, a fact which is not guaranteed in general. This is why a lot of effort has been made in analyzing *quantifier-free* interpolation: given two quantifier-free formulæ A and B such that $A \wedge B$ is not satisfiable (modulo a theory T), a *quantifier-free interpolant* C is a quantifier-free formula such that $T \models A \to C, T \models C \wedge B \to \bot$ and such that C contains only the variables which occur both in A and in B.

The specific theories we consider here are variants of *McCarthy's theory of arrays* [4] *with extensionality.* This theory is suitable to formalize programs over arrays, like standard copying,

SMT 21: 19th International Workshop on Satisfiability Modulo Theories, 18–19 July 2021, online
jose.castellanosjoo@cs.unm.edu (J. A. Castellanos Joo); silvio.ghilardi@unimi.it (S. Ghilardi); gianola@inf.unibz.it (A. Gianola); kapur@cs.unm.edu (D. Kapur)

comparing, searching, sorting functions; verification problems of this kind are collected, e.g., in the SV-COMP benchmarks categories "ReachSafety-Arrays" and "MemSafety-Arrays"[1], where safety verification tasks involving arrays of *finite but unknown length* are considered.

By itself, the theory of arrays with extensionality does not have quantifier free interpolation [5]; however, in [6] it was shown that quantifier-free interpolation is restored if one enriches the language with a binary function skolemizing the extensionality axiom. Such a Skolem function, applied to two array variables a, b, returns an index $\mathtt{diff}(a, b)$ where a, b differ (it returns an arbitrary value if a is equal to b). This semantics for the `diff` operation is very undetermined and does not have a significant interpretation in concrete programs. That is why in [7] the `diff` operation is modified so as to obtain a defined and natural meaning: we introduce the theory of arrays with maxdiff $\mathcal{ARD}(T_I)$, where $\mathtt{diff}(a, b)$ returns the *biggest* index where a, b differ (in case $a = b$ we ask for $\mathtt{diff}(a, b)$ to be the minimum index 0). Since it is natural to view arrays as functions defined on initial intervals of the nonnegative integers, this choice has a clear semantic motivation. The expressive power of the theory of arrays so enriched becomes bigger: for instance, if we also add to the language a constant symbol ϵ for the undefined array constantly equal to some 'undefined' value $\perp$ (where $\perp$ is meant to be different from the values $a[i]$ actually in use), then we can define $|a|$ as $\mathtt{diff}(a, \epsilon)$. In this way we can model the fact that a is undefined outside the interval $[\,0, |a|\,]$ - this is useful to formalize the above mentioned SV-COMP benchmarks. The effectiveness of quantifier-free interpolation in the theory of arrays with maxdiff is exemplified in the `Strcpy` function example of Figure 1 in [7]:[2] we need to express $|\cdot|$ to formalize its initial states without quantifiers. In [7], $\mathcal{ARD}(T_I)$ is shown to admit quantifier-free interpolation, and a suitable algorithm based on instantiation à-la-Herbrand and on a hierarchical approach ([8, 9, 10]) is introduced: by exploiting iterated applications of the maxdiff operator and reducing the interpolation problem to a reduced interpolating theory, we can compute quantifier-free interpolants out of the input formulæ.

In this paper, we present (in Section 3) AXDInterpolator[3], a tool that implements the interpolation algorithm of [7]. Following an instantiation-based approach, the algorithm *always* (and in particular for all the examples in this paper) returns a quantifier-free formula. A comparison is done with the state-of-the-art interpolation engines that have known interpolant generation algorithms for the theory of arrays–`iZ3`, `MathSat`, and `SMTInterpol` [11, 12, 13] support the theory of arrays with maxdiff. Indeed, on examples supporting expressive features like `diff` or the $|\cdot|$ operator, no proper comparison is possible. In case unsafe traces for such examples are considered, in contrast to AXDInterpolator, `Mathsat` and `iZ3` cannot usually get a quantifier-free interpolant: either the tools detect the satisfiability of the unsafe trace (since they reason modulo a theory that is *weaker* than $\mathcal{ARD}(T_I)$) so they cannot compute an interpolant,

[1] https://sv-comp.sosy-lab.org/2020/benchmarks.php

[2] The `Strcpy` function performs N sequential index-wise copies of values of two arrays a, b from a to b where N is the length of both arrays: the loop invariant for this examples is $a = b \vee (N > \mathtt{diff}(a, b) \wedge \mathtt{diff}(a, b) \geq I)$, I being the index variable in the loop. This formula is a loop invariant because if the two arrays a, b are different while $I < N$ then the conjunction $N > \mathtt{diff}(a, b) \wedge \mathtt{diff}(a, b) \geq I$ asserts that the possibly entry where a, b differ has not been processed yet.

[3] We use the prerelease version 0.9.0 of our tool: https://github.com/typesAreSpaces/AXDInterpolator/releases/tag/0.9.0

or, in case the unsafe trace is correctly detected as unsat (equipping the solver with axioms for maxdiff), they fail returning errors or *quantified* interpolant or timeout; for Example 1 from [7], `SMTInterpol` fails. `Mathsat` and `iZ3` fail also in the case of the example from [5], which is well-known not to admit a quantifier-free interpolant unless the extensionality axiom is skolemized with a generic diff operator: in this specific case, `SMTInterpol`, instead, can successfully compute a quantifier-free interpolant using the plain diff operator, but AXDInterpolator can compute an interpolant with maxdiff a bit faster. Running experiments over two tracks of the SV-COMP (ReachSafety-Arrays and MemSafety-Arrays), we use unsatisfiable formulæ extracted with the model checker UAutomizer [14]: the latter automatically transforms C programs taken from those benchmarks into suitable SMTLIB2 files that allow to verify particular properties of these programs. Since UAutomizer does not support the extended language of arrays with maxdiff, the comparison among the different solvers becomes acceptable. The results show that AXDInterpolator, `iZ3`, `Mathsat` and `SMTInterpol` have an overall identical behavior over these benchmarks: interpolants are computed for almost every example, apart from a small portion of timeout outcomes reached by AXDInterpolator as its implementation is evolving. On more expressive examples, such as Example 1 from [7], other solvers including the most expressive `SMTInterpol` fail, whereas AXDinterpolator produces useful interpolants. AXD-Interpolator uses the `Z3` API [15] and allows the user to choose among different solvers in order to compute the interpolants in the reduced theory invoked by the hierarchical method: `iZ3`, `Mathsat` [12] and `SMTInterpol` [13]. It generates a formula in SMTLIB2 format, which guarantees compatibility with model checkers and invariant generators adopting the same format.

2. Arrays with MaxDiff

The *McCarthy theory of arrays* [4] *with extensionality* has three sorts `ARRAY`, `ELEM`, `INDEX` (called "array", "element", and "index" sort, respectively) and two function symbols rd ("read") and wr ("write") of appropriate arities. Besides the axioms for defining rd and wr, it contains the 'extensionality' axiom: $\forall x, y.\ x \neq y \rightarrow (\exists i.\ rd(x, i) \neq rd(y, i))$. In [7], the Skolem function `diff` for this axiom has been given a proper semantics, i.e., `diff` returns the biggest index where two different arrays differ. For introducing our theory of arrays with maxdiff, we first need to consider an *index theory* T_I (see [7] for the general definition). The signature of an index theory T_I contains at least the binary relation symbol $\leq$ and the constant 0. Examples of index theories are the theory of total orders TO, integer difference logic $\mathcal{IDL}$, integer linear arithmetic $\mathcal{LIA}$, and real linear arithmetics $\mathcal{LRA}$. For most applications, $\mathcal{IDL}$ (namely the theory of integer numbers with 0, ordering, successor and predecessor) suffices as in this theory one can model counters for scanning arrays.

Given an index theory T_I, we now introduce our *array theory with maxdiff* $\mathcal{ARD}(T_I)$ (parameterized by T_I) as follows. We still have the same sorts; the language includes the symbols of T_I, the operations rd, wr, a binary function `diff` of type `ARRAY` $\times$ `ARRAY` $\rightarrow$ `INDEX`, as well as constants ϵ and $\perp$ of sorts `ARRAY` and `ELEM`, resp. The constant $\perp$ models an undetermined (e.g. undefined, not-in-use, not coming from appropriate initialization, etc.) value and ε models the totally undefined array; the term $\texttt{diff}(x, y)$ returns the maximum index where x and y

differ and returns 0 if x and y are equal.[4] Formally, the axioms of $\mathcal{ARD}(T_I)$ include, besides the axioms of T_I, the following ones:

$$\forall y, i, e. \qquad i \geq 0 \to rd(wr(y,i,e),i) = e \tag{1}$$
$$\forall y, i, j, e. \qquad i \neq j \to rd(wr(y,i,e),j) = rd(y,j) \tag{2}$$
$$\forall x, y. \qquad x \neq y \to rd(x, \mathtt{diff}(x,y)) \neq rd(y, \mathtt{diff}(x,y)) \tag{3}$$
$$\forall x, y, i. \qquad i > \mathtt{diff}(x,y) \to rd(x,i) = rd(y,i) \tag{4}$$
$$\forall x. \qquad \mathtt{diff}(x,x) = 0 \tag{5}$$
$$\forall x.i \qquad i < 0 \to rd(x,i) = \bot \tag{6}$$
$$\forall i. \qquad rd(\varepsilon, i) = \bot \tag{7}$$

As an effect of the above axioms, we have that an array x is undefined outside the interval $[0, |x|]$, where $|x|$ is defined as $|x| := \mathtt{diff}(x, \varepsilon)$. Typically, this interval is finite, and it can be shown that any satisfiable constraint is satisfiable in a model where all such intervals (involving variables appearing in the constraint) are finite [7].

For the interpolation algorithm from [7], we need to introduce iterated `diff` operations, similarly to [10]. As we know, $\mathtt{diff}(a,b)$ returns the biggest index where a and b differ (it returns 0 if $a = b$). Now we want an operator that returns the last-but-one index where a, b differ (0 if a, b differ in at most one index), etc. Our language is already enough expressive for that, so we can introduce such operators explicitly as follows. Given array variables a, b, we define by mutual recursion the sequence of array terms $b_1, b_2, \ldots$ and of index terms $\mathtt{diff}_1(a,b), \mathtt{diff}_2(a,b), \ldots$:

$$b_1 := b; \qquad \mathtt{diff}_1(a,b) := \mathtt{diff}(a, b_1);$$
$$b_{k+1} := wr(b_k, \mathtt{diff}_k(a,b), rd(a, \mathtt{diff}_k(a,b))); \qquad \mathtt{diff}_{k+1}(a,b) := \mathtt{diff}(a, b_{k+1})$$

Lemma 2.1. *A formula like*

$$\mathtt{diff}_1(a,b) = k_1 \wedge \cdots\cdots \wedge \mathtt{diff}_l(a,b) = k_l \tag{8}$$

is equivalent modulo $\mathcal{ARD}$ to the conjunction of the following five formulæ:

$$k_1 \geq k_2 \wedge \cdots \wedge k_{l-1} \geq k_l \wedge k_l \geq 0 \tag{9}$$
$$\textstyle\bigwedge_{j<l} (k_j > k_{j+1} \to rd(a,k_j) \neq rd(b,k_j)) \tag{10}$$
$$\textstyle\bigwedge_{j<l} (k_j = k_{j+1} \to k_j = 0) \tag{11}$$
$$\textstyle\bigwedge_{j\leq l} (rd(a,k_j) = rd(b,k_j) \to k_j = 0) \tag{12}$$
$$\forall h \; (h > k_l \to rd(a,h) = rd(b,h) \vee h = k_1 \vee \cdots \vee h = k_{l-1}) \tag{13}$$

Separated Pairs, M-Instantiation and Satisfiability. The key step of the interpolation algorithm from [7] depends upon the problem of checking satisfiability (modulo $\mathcal{ARD}(T_I)$) of quantifier-free formulæ; this is solved by adapting instantiation techniques, like those from [16].

[4] Notice that it might well be the case that $\mathtt{diff}(x,y) = 0$ for different x, y, but in that case 0 is the only index where x, y differ.

The *complexity* $c(t)$ of a term t is defined as the number of function symbols occurring in t. If $\mathcal{I}$ is a set of T_I-terms, an $\mathcal{I}$*-instance* of a universal formula of the kind $\forall i\, \phi$ is a formula of the kind $\phi(t/i)$ for some $t \in \mathcal{I}$.

A pair of sets of quantifier-free formulæ $\Phi = (\Phi_1, \Phi_2)$ is a *separated pair* iff

(1) Φ_1 contains equalities of the form $\mathtt{diff}_k(a, b) = i$ and $a = wr(b, i, e)$; moreover if it contains the equality $\mathtt{diff}_k(a, b) = i$, it must also contain an equality of the form $\mathtt{diff}_l(a, b) = j$ for every $l < k$;

(2) Φ_2 contains Boolean combinations of T_I-atoms and of atoms of the forms: $\{rd(a, i) = rd(b, j),\ \ rd(a, i) = e,\ \ e_1 = e_2\}$, where a, b, i, j, e, e_1, e_2 are variables or constants of the appropriate sorts.

The separated pair is said to be finite iff Φ_1 and Φ_2 are both finite. In practice, in a separated pair $\Phi = (\Phi_1, \Phi_2)$, reading $rd(a, i)$ as a functional application, it turns out that *the formulæ from* Φ_2 *can be translated into quantifier-free formulæ of the combined theory* $T_I \cup \mathcal{EUF}$ (the array variables occurring in Φ_2 are converted into free unary function symbols). $T_I \cup \mathcal{EUF}$ enjoys the decidability of the quantifier-free fragment and has quantifier-free interpolation because T_I is an index theory [7]: we adopt a hierarchical approach (similar to [8, 9]) and *we rely on satisfiability and interpolation algorithms for such a theory as black boxes.*

Let $\mathcal{I}$ be a set of T_I-terms and let $\Phi = (\Phi_1, \Phi_2)$ be a separated pair; we let $\Phi(\mathcal{I}) = (\Phi_1(\mathcal{I}), \Phi_2(\mathcal{I}))$ be the smallest separated pair satisfying the following conditions:

- $\Phi_1(\mathcal{I})$ is equal to Φ_1 and $\Phi_2(\mathcal{I})$ contains Φ_2;
- $\Phi_2(\mathcal{I})$ contains all $\mathcal{I}$-instances of the two formulæ $\forall i\ \ rd(\varepsilon, i) = \bot$, $\forall i\, (i < 0 \to rd(a, i) = \bot)$, where a is any array variable occurring in Φ_1 or Φ_2;
- if Φ_1 contains $a = wr(b, i, e)$, then $\Phi_2(\mathcal{I})$ contains *all the* $\mathcal{I}$*-instances of the equivalent formula* $(i \geq 0 \to rd(a, i) = e) \wedge \forall h\ (h \neq i \to rd(a, h) = rd(b, h))$;
- if Φ_1 contains the conjunction $\bigwedge_{i=1}^{l} \mathtt{diff}_i(a, b) = k_i$, then $\Phi_2(\mathcal{I})$ contains the formulæ (9), (10), (11), (12) as well as *all* $\mathcal{I}$*-instances of the formula* (13).

For $M \in \mathbb{N} \cup \{\infty\}$, the M*-instantiation* of $\Phi = (\Phi_1, \Phi_2)$ is the separated pair $\Phi(\mathcal{I}_\Phi^M) = (\Phi_1(\mathcal{I}_\Phi^M), \Phi_2(\mathcal{I}_\Phi^M))$, where $\mathcal{I}_\Phi^M$ is the set of T_I-terms of complexity at most M built up from the index variables occurring in Φ_1, Φ_2. The *full instantiation* of $\Phi = (\Phi_1, \Phi_2)$ is the separated pair $\Phi(\mathcal{I}_\Phi^\infty) = (\Phi_1(\mathcal{I}_\Phi^\infty), \Phi_2(\mathcal{I}_\Phi^\infty))$ (which is usually not finite). A separated pair $\Phi = (\Phi_1, \Phi_2)$ is M*-instantiated* iff $\Phi = \Phi(\mathcal{I}_\Phi^M)$; it is $\mathcal{ARD}(T_I)$-satisfiable iff so it is the formula $\bigwedge \Phi_1 \wedge \bigwedge \Phi_2$. The $SMT(\mathcal{ARD}(T_I))$ problem is decidable for every index theory T_I (and can be reduced to $T_I \cup \mathcal{EUF}$-satisfiability) thanks to the following results:

Lemma 2.2. *Given a quantifier-free formula ϕ, we can compute finitely many finite separation pairs $\Phi^1 = (\Phi_1^1, \Phi_2^1), \ldots, \Phi^n = (\Phi_1^n, \Phi_2^n)$ such that ϕ is $\mathcal{ARD}(T_I)$-satisfiable iff so is one of the Φ^i.*

Lemma 2.3. *The following conditions are equivalent for a finite separation pair $\Phi = (\Phi_1, \Phi_2)$:* (i) Φ *is $\mathcal{ARD}(T_I)$-satisfiable;* (ii) $\bigwedge \Phi_2(\mathcal{I}_\Phi^0)$ *is $T_I \cup \mathcal{EUF}$-satisfiable.*

2.1. The interpolation algorithm

The interpolation algorithm from [7] is based on instantiation à-la-Herbrand. Our problem is to compute a *quantifier-free interpolant* in $\mathcal{ARD}(T_I)$ of two quantifier-free formulæ A and B.

We call the variables occurring in both A and B *common variables*, whereas the variables occurring in A (resp. in B) are called *A-variables* (resp. *B-variables*). The same terminology applies to terms, atoms and formulæ. As shown in [7], both A and B can be given in the form of finite separated pairs. Thus, A is of the form $\bigwedge A_1 \wedge \bigwedge A_2$, B is of the form $\bigwedge B_1 \wedge \bigwedge B_2$, for separated pairs (A_1, A_2) and (B_1, B_2). Also, A and B can be assumed to be both 0-instantiated.

The formulæ from A_2 and B_2 are formulæ from the signature of $T_I \cup \mathcal{EUF}$ (after rewriting terms of the kind $rd(a, i)$ to $f_a(i)$, where the f_a are free function symbols). Of course, if $A_2 \wedge B_2$ is $T_I \cup \mathcal{EUF}$-inconsistent, *we can get our quantifier-free interpolant by using our black box algorithm for interpolation in the weaker theory $T_I \cup \mathcal{EUF}$.*

Formally, we *apply the Loop below until $A_2 \wedge B_2$ becomes $T_I \cup \mathcal{EUF}$-inconsistent*: Theorem 4 in [7] guarantees that, if $A \wedge B$ is $\mathcal{ARD}(T_I)$-inconsistent, then the Loop always terminates. When $A_2 \wedge B_2$ becomes inconsistent, we can get our interpolant via the interpolation algorithm for $T_I \cup \mathcal{EUF}$. We need a counter M recording how many times the Loop has been executed.

Loop *(to be repeated until $A_2 \wedge B_2$ becomes inconsistent modulo $T_I \cup \mathcal{EUF}$, initially $M = 0$). Pick two distinct common* `ARRAY`*-variables c_1, c_2 and $n \geq 1$ and such that no conjunct of the kind $\mathtt{diff}_n(c_1, c_2) = k$ occurs in both A_1 and B_1 for some $n \geq 1$ (but such that for every $l < n$ there is a conjunct of the form $\mathtt{diff}_l(a, b) = k$ occurring in both A_1 and B_1). Pick also a fresh* `INDEX` *constant k_n; conjoin $\mathtt{diff}_n(c_1, c_2) = k_n$ to both A_1 and B_1; then M-instantiate both A and B. Increase M to $M + 1$.*

Notice that the fresh index constants k_n introduced during the Loop are considered common constants and so they are considered in the M-instantiation of both A and B. In the output, these constants k_n need to be eliminated.

3. Implementation

Our implementation is written in C++ using the `Z3` C++ API [15]. It takes as input:

- A SMTLIB2 file using standard elements of the language.
- An *index theory* T_I [7]. Currently, we support quantifier free fragments of total order theory (`QF_TO`), integer difference (`QF_IDL`), and linear integer arithmetic (`QF_LIA`).
- A positive integer indicating a maximum number of allowed iterations of the main loop. [5]
- An integer number denoting the SMT solver to use to compute the interpolant in the case the input is unsatisfiable in the reduced theory $T_I \cup \mathcal{EUF}$. Currently, we support the SMT solvers `Z3`, `Mathsat`, and `SMTInterpol`.

Our extended language is parameterized by the array sorts in the input formula as well as by an *index theory*. The domain sort of every array is currently implemented using the `Int` sort [6].

```
(declare-sort A)
(declare-fun diff'A' ((Array Int A) (Array Int A)) Int)
(declare-fun length'A' ((Array Int A)) Int)
(declare-fun empty_array'A' () (Array Int A)
(declare-fun undefined'A' () A)
```

[5] Index theories stronger than `QF_TO` require such upper bound for termination purposes in reasonable time.

[6] The requirement of including additional information on the name of sorted-functions in our extended language is necessary since `Mathsat` does not implement *ad hoc* polymorphism.

The rest of the SMTLIB2 file should specify two assertions. The first one encodes the `A-part` and the second one the `B-part` of the interpolation problem. In the following we describe the algorithm in [7] by phases as it was implemented in our tool.

Preprocessing. This step takes two main parts. First, our tool scans the declaration section in the input file to internalize the signature involving the sorts of the array elements present in the input file. Additionally, we instantiate new declarations of uninterpreted sorts `ArraySortA` which *abstract* the array sorts present in the input file, as well as two new functions:

```
(declare-fun wr ((ArraySortA Int A)) ArraySortA)
(declare-fun rd (ArraySortA Int) A)
```

with a proper uninterpreted sort. The reason for this is to internalize an uninterpreted language in the algorithm in order to perform theory reduction from $\mathcal{ARD}(T_I)$ to $T_I \cup \mathcal{EUF}$.

The second part involves processing the input formula. The algorithm in [7] handles a conjunctive fragment situation. Thus, the algorithm uses `Z3` tactics to perform a DNF transformation and produce disjuncts of conjunctions. Every disjunct is processed by the `AXDInterpolator` C++ class in our implementation which can handle conjunctive formulæ. Each conjunctive formula aforementioned is normalized (we replace `length'A'(x)` subformulæ for `diff'A'(x, empty-array'A')`) and flattened by introducing constants of the right sort. During the execution of the aforementioned methods, we keep track of the Int constants as well as array constants appearing in the input formula. Index constants are used to instantiate quantified formulæ of the respective $\mathcal{ARD}(T_I)$ theory in use while array constants are necessary to compute common pairs of arrays between the `A-part` and the `B-part` of the interpolation problem in the main loop of the algorithm.

Standard Pairs. Separated pairs from Section 2 are called in our implementation *Standard Pairs*, and their requirements are encoded as a C++-class called `StandardInput`. There, atoms $\{rd(a, b) = rd(b, j), rd(a, i) = e, e_1 = e_2\}$ are such that a, b are of sort `ArraySort'A'`, i, j are of sort `Int`, e is of uninterpreted sort `A`, and e_1, e_2 are of appropriate sorts.

Given a formula, this class keeps two separate `z3::expr_vector` denoting the conjunction of formulæ on the first and second component of the standard pair. We use one instantiation of the `StandardInput` class for both formulæ `A-part` and `B-part`. Additionally, this class implements methods handling the instantiations of formulæ. Instantiation is achieved by including an undefined `z3::expr` which is replaced using the `z3.substitute(z3::expr_vector const &, z3::expr_vector const&)` method.

Main Loop of the Algorithm and Interpolation Generation. First, the algorithm identifies the set of common array constants between `A-part` and the `B-part`. If this set is empty or contains a single element, the algorithm proceeds to instantiate an `z3::solver` and check the satisfiability of the second component of the Standard Pairs of the `A-part` and `B-part`. Otherwise, it uses an iterator to select common pair of array constants which selects elements in a fair way [7]. Similarly to the previous case, it checks the satisfiability of the second component

[7]A selection over n objects can be seen as a function $f : \{1, \ldots, n\} \to \mathbb{N}$. If $i \in \{1, \ldots, n\}$, $f(i)$ denotes the

of the Standard Pairs. If the state of the solver is unsatisfiable the loop terminates saving the formulæ of the Standard Pairs of the A-part and B-part. Otherwise, it selects the common pair (c_1, c_2) of array constants and 'propagates' this pair to each of the Standard Pairs of the A-part and B-part. The propagation consists of adding the equation of the form $\mathtt{diff}_n(c_1, c_2) = k$ to the first component of the Standard Pairs where n is the smallest integer number such that $\mathtt{diff}_n(c_1, c_2)$ does not occur in the Stand Pair of both A-part and B-part and k is a fresh Int constant. Subsequently, both Standard Pairs are $N - instantiated.$

If the *index theory* in use extends QF_TO, then the implementation uses heuristics to increase the value of an internal number keeping track of the maximum allowed complexity on the index terms. We noticed the implementation performs well if only $0 - instantiations$ are executed. Better heuristics are under investigation.

If in the main loop the solver eventually detects that the second component of the Standard Pairs is unsatisfiable, then the implementation saves the formulæ of the Standard Pairs of the A-part and B-part respectively. Depending on the choice of the user, the implementation executes a command on an external shell calling the SMT-solver to compute the interpolant using the pair of unsatisfiable formulæ mentioned before. For the latter, the implementation 'compiles' a temporary SMTLIB2 file for the respective SMT-solver chosen.

Reading the interpolant result from the previous process the algorithm removes added constants in order to provide a solution using elements from the input formula. Since no SMT-solver supports interpolation algorithms for the QF_TO we select the QF_IDL as the theory for the interpolation engine and use a custom rewriter implemented on Z3 in order to relax a QF_IDL formula as a QF_TO statement. The latter is important for us in order to remain closed in the respective language.

Correctness of the Output. If the interpolant is successfully computed, our implementation allows the user to automatically verify the correctness of the output, that is the output is a real interpolant in $\mathcal{ARD}(T_I)$: this is done by exploiting the first-order axioms of $\mathcal{ARD}(T_I)$ and by calling a z3::solver instance using them. For the latter the implementation 'lifts' the result by replacing applications of $\mathtt{diff}_k$ to their recursive definition. This is achieved using a data structure that memorizes these intermediate formulæ.

4. Evaluation

4.1. Some interesting examples

We present in detail an example of two formulæ A, B such that our implementation can compute a quantifier-free interpolant for them (in 0.066 sec), whereas the other solvers cannot, since: (i) modulo the standard theory of arrays, they return that $A \wedge B$ is sat; (ii) when enriched with the quantified axioms of $\mathcal{ARD}(T_I)$, iZ3 cannot return an answer within 15 minutes, SMTInterpol correctly detects that $A \wedge B$ is unsat but cannot return an interpolant and Mathsat returns error since it does not support quantifiers. The example is taken from [7] (Example 1):

number of times the object i has been selected. A selection over n objects is *fair* if for all i, j in $\{1, \ldots, n\}$ we have that $|f(i) - f(j)| \leq 1$.

- A-part: $\texttt{diff}(a, c_1) = i_1 \wedge \texttt{diff}(b, c_2) = i_1 \wedge a = wr(a_1, i_3, e_3) \wedge a1 = wr(b, i_1, e_1)$
- B-part: $i_1 < i_2 \wedge i_2 < i_3 \wedge rd(c_1, i_2) \neq rd(c_2, i_2)$

Our implementation returns the following as an interpolant (and verifies its correctness):

- Interpolant: let $a!1 = \neg(rd(c_2, \texttt{diff}(c_2, c_1)) = rd(c_1, \texttt{diff}(c_2, c_1)))$ be in let $a!2 = \neg(\texttt{diff}(c_2, c_1) \leq i_1) \wedge a!1 \wedge \neg(i_3 \leq \texttt{diff}(c_2, c_1))$ be in $\texttt{diff}_2(c_2, c_1) \leq i_1 \wedge \neg a!2$

If the previous file is given the `iZ3`, `SMTInterpol` and `Mathsat`, then the SMT-solvers output sat: clearly, the axioms for maxdiff are missing and $A \wedge B$ is satisfiable in the standard theory of arrays. Only AXDInterpolator is able to compute a quantifier-free interpolant.

Consider this example from [5]: $A := x = wr(y, i, e)$ and $B := rd(x, j) \neq rd(y, j) \wedge rd(x, k) \neq rd(y, k) \wedge j \neq k$. When adding the axioms for maxdiff, `Mathsat` and `iZ3` are not able to find an interpolant, whereas both `SMTInterpol` and AXDInterpolator get a quantifier-free interpolant: AXDInterpolator is a bit faster than `SMTInterpol` (0.062 vs 0.176 sec), and the output of the latter uses the plain diff with the undetermined semantics. The `Strcpy` function from [7] gives the same results, and AXDInterpolator is still a bit faster than `SMTInterpol` (0.057 vs 0.205 sec).

4.2. Benchmarks using SV-COMP and UAutomizer

We also tested our implementation on two benchmarks. We performed our experiments on a machine with Linux 5.11.16, Intel i7-9700 4.7 GHz with 32 GB of memory. We used the model checker UAutomizer [14] to extract their SMT Scripts which are SMTLIB2 files containing the queries produced in order to verify some particular property of C programs. In our case, we chose C-programs from the ReachSafety-Arrays and MemSafety-Arrays tracks of the SV-COMP [17]. We let the machine produce SMT Scripts for 15 minutes. We used these SMT Scripts files to compare the number of interpolants computed from unsatisfiable formulæ. For the latter we assigned each process up to 360 seconds and 6 GB of memory [8]. All the examples do not include the expressive features of $\mathcal{ARD}(T_I)$, since they have been automatically produced by UAutomizer: this allows us to run these examples on the other solvers `iZ3`, `Mathsat` and `SMTInterpol`. We defined two outcomes: Success, and Timeout. Success is declared when an interpolant is computed, and Timeout when the process takes more time or space than the previously specified. No file failed (i.e., terminated without returning an interpolant). Our tool was configured to run with `QF_LIA` and 1000 allowed loop repetitions with the heuristic of only `0-instantiations`. The solvers were configured to use the `QF_AUFLIA` theory. The results are shown in Tables 1,2,3 and 4. The implementation is available at the following link: https://github.com/typesAreSpaces/AXDInterpolator. The experimental results can be obtained following the instructions in the documentation.

[8]We were able to use 1008 unsatisfiable formulæ from the ReachSafety-Arrays track; 110 queries contained array constants or function applications of select/store. From the MemSafety-Arrays track, we were able to obtain 758 unsatisfiable formulæ; 750 contained array related elements. We noticed the input formulæ do not contain applications of any diff operator as expected since the proposed extension of $\mathcal{ARD}(T_I)$ is relatively new.

Subtracks	AXD Interpolator iZ3		Mathsat		SMTInterpol	
	Success	Timeout	Success	Timeout	Success	Timeout
array-examples	584	1	584	1	584	1
array-memsafety	118	0	118	0	118	0
termination-crafted	52	3	52	3	52	3

Table 1
Memsafety-track results - Our implementation

Subtracks	iZ3		Mathsat		SMTInterpol	
	Success	Timeout	Success	Timeout	Success	Timeout
array-examples	585	0	585	0	585	0
array-memsafety	118	0	118	0	118	0
termination-crafted	55	0	55	0	55	0

Table 2
Memsafety-track results - Other Solvers

Subtracks	AXD Interpolator iZ3		Mathsat		SMTInterpol	
	Success	Timeout	Success	Timeout	Success	Timeout
array-cav19	31	0	31	0	31	0
array-examples	50	0	50	0	50	0
array-fpi	774	21	774	21	774	21
array-industry-pattern	8	0	8	0	8	0
array-lopstr16	54	0	54	0	54	0
array-patterns	11	0	11	0	11	0
array-tiling	6	0	6	0	6	0
reducercommutativity	53	0	53	0	53	0

Table 3
Reachsafety-track results - Our implementation

Subtracks	iZ3		Mathsat		SMTInterpol	
	Success	Timeout	Success	Timeout	Success	Timeout
array-cav19	31	0	31	0	31	0
array-examples	50	0	50	0	50	0
array-fpi	795	0	795	0	795	0
array-industry-pattern	8	0	8	0	8	0
array-lopstr16	54	0	54	0	54	0
array-patterns	11	0	11	0	11	0
array-tiling	6	0	6	0	6	0
reducercommutativity	53	0	53	0	53	0

Table 4
Reachsafety-track results - Other Solvers

5. Related Work

In [11] McMillan introduced an interpolating proof calculus to extract interpolants using refutational proofs obtained from the `Z3` SMT-solver. His approach benefits from the flexibility of `Z3` to handle a variety of theories and their combination: it relies on a secondary interpolation solver in order to 'fill the gaps' of refutational proofs introduced by *theory lemmas*, which are formulæ derived by the satellities theories encoded in `Z3` without a detailed explanation in the proof. The secondary interpolation solver only requires an interpolation algorithm for `QF_UFLIA`. Since the theory of arrays is introduced using quantified formulæ, this approach generates quantified formulæ.

The authors in [18] employed the proof tree preserving interpolation scheme from [19] to compute interpolants using a resolution proof. This approach is capable of handling mixed literals but not mixed terms[9], so the authors introduced the idea of *weakly equivalences* between arrays to handle those cases for their interpolation calculi. The authors include the `diff` operation between arrays in order to compute quantifier-free interpolants, but no additionally semantical properties are given to the `diff` operation.

The authors in [10] proposed a general framework for computing interpolants via reduction using finite instantiations. In particular, the paper discusses complete interpolation procedures for theories of arrays and linked lists, as well as other heap-allocated data structures.

6. Conclusions

In this paper we presented AXDInterpolator, the implementation of the interpolation algorithm from [7]. We showed the feasibility of AXDInterpolator by validating it on two benchmarks from the SV-COMP. We also compared our implementation with state-of-the-art solvers: apart from very few timeout outcomes, our tool handled all the examples the other solvers did. We also managed to handle interesting examples that are not handled by the other solvers, which makes the option of our tool appealing. Specifically, expressive examples that make use of features like $|\cdot|$ cannot in general be managed by any existing solver: this suggests that AXDInterpolator could be applied for solving more sophisticated examples coming from concrete C programs using arrays. This analysis is a significant point to investigate in future work.

We noticed that our implementation, in case of the SV-COMP benchmarks, is behind in time with respect to the state-of-the-art solvers. We think the main reasons might be the following:

- The `UAutomizer` model checker does not include our extended signature in their SMT Scripts. We might take into consideration that many industrial problems will not explicitly use this language, however the overall structure of many array programs commonly use $|\cdot|$ for various algorithms. Thus, we need to integrate this structure in the input formula.
- Our implementation timeouts slightly more often than other solvers. This point can be further improved by doing better engineering work at the implementation level. The DNF conversion (in worst case, exponential) appears to be the first target to rework.
- The current design does not perform incremental satisfiability checks. As future work we plan to perform incremental checks since this can improve the performance of the used `z3::solver` data structure. Incremental checks can be implemented due to the incremental nature of the proposed interpolation algorithm by including a hash consed data structure on the terms/predicates produced in the main loop of the algorithm and because the data structure `z3::solver` can keep track of previously proven assertions.

References

[1] K. L. McMillan, Interpolation and SAT-based model checking, in: Proc. of CAV, volume 2725 of *LNCS*, Springer, 2003, pp. 1–13. doi:10.1007/978-3-540-45069-6_1.

[9] Given an interpolation pair (A, B), a symbol s is *A-local* if s appears in the symbols of A but not in the symbols of B. A literal/term is mixed if it contains *A-local* and *B-local* symbols

[2] K. L. McMillan, Lazy abstraction with interpolants, in: Proc. of CAV, volume 4144 of *LNCS*, Springer, 2006, pp. 123–136. doi:10.1007/11817963_14.
[3] W. Craig, Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory, J. Symbolic Logic 22 (1957) 269–285.
[4] J. Mccarthy, Towards a mathematical science of computation, in: In IFIP Congress, North-Holland, 1962, pp. 21–28.
[5] D. Kapur, R. Majumdar, C. G. Zarba, Interpolation for Data Structures, in: Proc. of SIGSOFT-FSE, ACM, 2006, pp. 105–116.
[6] R. Bruttomesso, S. Ghilardi, S. Ranise, Quantifier-free interpolation of a theory of arrays, Log. Methods Comput. Sci. 8 (2012).
[7] S. Ghilardi, A. Gianola, D. Kapur, Interpolation and amalgamation for arrays with maxdiff, in: S. Kiefer, C. Tasson (Eds.), Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, volume 12650 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 268–288.
[8] V. Sofronie-Stokkermans, Interpolation in local theory extensions, Log. Methods Comput. Sci. 4 (2008).
[9] V. Sofronie-Stokkermans, On interpolation and symbol elimination in theory extensions, Log. Methods Comput. Sci. 14 (2018).
[10] N. Totla, T. Wies, Complete instantiation-based interpolation, J. Autom. Reasoning 57 (2016) 37–65.
[11] K. L. McMillan, Interpolants from Z3 proofs, in: Proc. of FMCAD, 2011, pp. 19–27.
[12] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, R. Sebastiani, MathSAT: Tight integration of SAT and mathematical decision procedures, Journal of Automated Reasoning (2005).
[13] J. Christ, J. Hoenicke, A. Nutz, SMTInterpol: An interpolating SMT solver, in: A. F. Donaldson, D. Parker (Eds.), Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings, volume 7385 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 248–254.
[14] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, A. Podelski, Ultimate Automizer with SMTInterpol, in: N. Piterman, S. A. Smolka (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 641–643.
[15] L. De Moura, N. Bjørner, Z3: An Efficient SMT Solver, in: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 337–340.
[16] A. R. Bradley, Z. Manna, H. B. Sipma, What's decidable about arrays?, in: Proc. of VMCAI, volume 3855 of *LNCS*, Springer, 2006, pp. 427–442. doi:10.1007/11609773_28.
[17] J. F. Groote, K. G. Larsen, D. Beyer, Software verification: 10th comparative evaluation (sv-comp 2021), Tools and Algorithms for the Construction and Analysis of Systems27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March

27 – April 1, 2021, Proceedings, Part II 12652 (2021) 401—422. URL: https://europepmc.org/articles/PMC7984550.

[18] J. Hoenicke, T. Schindler, Efficient interpolation for the theory of arrays, in: Proc. of IJCAR, volume 10900 of *LNCS (LNAI)*, Springer, 2018, pp. 549–565. doi:10.1007/978-3-319-94205-6_36.

[19] J. Christ, J. Hoenicke, A. Nutz, Proof tree preserving interpolation, in: N. Piterman, S. A. Smolka (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 124–138.