Efficient and Complete FD-Solving for Extended Array Constraints*

Quentin Plazar[1], **Mathieu Acher**[1], **Sébastien Bardin** [2] and **Arnaud Gotlieb**[3]
[1] INRIA Bretagne-Atlantique, Rennes, France
[2] CEA, LIST, Gif-sur-Yvette, F-91191, France
[3] Simula Research Laboratory, Certus Center, Lysaker, Norway
{quentin.plazar, mathieu.acher}@inria.fr, sebastien.bardin@cea.fr, arnaud@simula.no

Abstract

Array constraints are essential for handling data structures in automated reasoning and software verification. Unfortunately, the use of a typical finite domain (FD) solver based on local consistency-based filtering has strong limitations when constraints on indexes are combined with constraints on array elements and size. This paper proposes an efficient and complete FD-solving technique for extended constraints over (possibly unbounded) arrays. We describe a simple but particularly powerful transformation for building an equisatisfiable formula that can be efficiently solved using standard FD reasoning over arrays, even in the unbounded case. Experiments show that the proposed solver significantly outperforms FD solvers, and successfully competes with the best SMT-solvers.

1 Introduction

Context. Automated reasoning and deduction systems are increasingly used in the domain of software verification, typically for checking the adequacy between program behavior and a formal specification. Numerous theories and powerful tools have been developed to prove different kinds of properties of a program [De Moura and Bjørner, 2008; Barrett et al., 2011]. The most popular approach consists in rewriting verification problems as satisfiability problems, that can then be solved using SMT solvers. Another path in the quest of an efficient and expressive solving is the use of Constraint Programming over Finite Domains (FD) [Bardin and Gotlieb, 2012; Bardin and Herrmann, 2008; Charreteur et al., 2009; Marre and Blanc, 2005]. FD has been proved useful for reasoning about complex structures crucial to software verification, such as floating point numbers [Botella et al., 2006], modular arithmetic [Gotlieb et al., 2010] and bitvectors [Bardin et al., 2010]. Multi-theory and verification-oriented FD frameworks are also starting to emerge [Marre and Blanc, 2005].

Our Problem. One aspect that is still lacking in constraint-based approaches to verification is an efficient handling of data structures. In many cases, constraints on such structures can be modeled using the theory of arrays [Kroening and Strichman, 2016]. One difficulty is that, unlike other theories (e.g., bitvectors, modular arithmetic), arrays do not provide many opportunities for efficient application of local-consistency based filtering techniques that are at the core of FD approaches. Instead, solving array constraints often requires reasoning on the global structure of formulas. Previous works [Hentenryck and Carillon, 1988; Charreteur et al., 2009; Bardin and Gotlieb, 2012] on this theory enabled to build FD tools to handle the fixed-size case, i.e., when arrays are known to have a fixed, finite size. They allowed to tackle some verification focused problems, such as test case generation. However, in many applications relevant to verification, such as unit verification, there is typically much fewer information available about data structure sizes. These sizes are sometimes known to be smaller than a (potentially big) given bound (bounded case), or no information is available about the size (unbounded case). *Overall, FD-based techniques extend poorly to the bounded case while the unbounded case is considered out of scope.*

Challenge and Goals. We tackle the problem of finding efficient FD-based techniques for reasoning about fixed-size, bounded and unbounded arrays, as typically found in verification-oriented problems. We pay particular attention to the following points:

- **Efficiency:** The overhead when dealing with large arrays should be reasonable;
- **Completeness:** Our technique should handle complex constraints (array equality, unbounded size) in a complete way;
- **Expressiveness:** It should be compatible with expressing other constraints on elements and indices, for example arithmetic constraints.

Contribution. **First**, we introduce array reduction, a formula transformation that takes as input a quantifier-free formula containing (possibly unbounded) arrays, and outputs an *equisatisfiable reduced formula using only fixed-sized arrays*. Thus the reduced formula can be solved using finite-domain solvers implementing array constraints [Gotlieb,

*Work partially funded by ANR (grant 14-CE28-0020).

Table 1: Handling extended array constraints

	CP	SMT	CP+R
fixed-size arrays	yes *size-dep*	yes *with N.O.*	yes
bounded-size arrays	yes *size-dep weak propag*	yes *with N.O.*	yes
unbounded arrays	no	yes	yes
arithmetic on indices	yes	yes *with N.O.*	yes

. CP+R: constraint programming, with our array reduction
. size-dep: complexity depend on array size
. N.O.: Nelson-Oppen solver combination [Nelson and Oppen, 1979] (expensive)

2009; Bardin and Gotlieb, 2012], with no modification to the propagators. Moreover, the sizes of the reduced arrays have no relation with the sizes of the arrays in the input formula, but only depend on the number of indices used in the input formula.

Second, we demonstrate that our encoding is efficient (Section 4.4), in the sense that the algorithm itself is cheap and that the obtained search space is significantly reduced w.r.t. alternative resolution methods for arrays – mainly because *our approach builds deeply on the many symmetries of the theory of arrays*.

Third, we implement this technique inside fdcc [Bardin and Gotlieb, 2012] and evaluate it (Section 5). We show that our technique allows one to solve previously unsupported formulas, typically over unbounded arrays. In addition, the resulting solver significantly outperforms FD reasoning in the case where all arrays have fixed sizes because of the powerful state space reduction opened during the transformation, and it also competes favorably with state-of-the art SMT solvers, especially when arrays have small sizes.

Overall, we achieve a promising trade-off between efficiency and expressiveness (see Table 1) with the native support of size/domains and handling of arithmetic on indices.

2 Motivation

Let us consider the following formula, where S is an integer constant, t is an array, and i, j are integers (indices):

$$\phi \triangleq size(t) = S \wedge t[i] < t[j] \wedge i = j$$

The formula ϕ has in fact no solution since $t[i] < t[j]$ implies that these two array accesses are different, while $i = j$ implies that they are equal. Constraint-based approaches typically rely on encoding the two array accesses $t[i]$ and $t[j]$ with ELEMENT constraints. However, the local filtering-based reasoning of constraint solvers is insufficient to detect this conflict. Therefore an exhaustive labeling will be required to conclude unsatisfiability of ϕ, essentially making the solving time dependent on the size S of the array.

For addressing such limitations, we introduce in this work a *formula transformation* that accounts for the fact that many array related conflicts are size independent, while preserving the ability to reason about sizes for arithmetic purposes. It consists of three main steps: separation of array literals from arithmetic literals in ϕ (purification), rewriting of size constraints as additional arithmetic constraints, and introduction of proxy variables to reason separately on array indices as array theory objects and arithmetic objects. The transformation eventually produces a reduced form for ϕ, denoted ϕ^r. An example is given hereafter. The two proxy variables are i^r and j^r. The renaming of t in t^r makes it clear that these two arrays are not identical. The consistency constraint is $i = j \Leftrightarrow i^r = j^r$.

$$\begin{array}{rll} \phi^r \triangleq & e < f \wedge i = j \wedge i, j \in 1..S & \wedge \\ & size(t^r) = 2 \wedge t^r[i^r] = e \wedge t^r[j^r] = f & \wedge \\ & i = j \Leftrightarrow i^r = j^r \wedge i^r = 1 \wedge j^r \in 1..2 & \end{array}$$

Note that S only appears in ϕ^r to define domains for i and j. Our transformation captures the idea that, since ϕ has only two array accesses (in i and j), considering an array with S elements is not useful – intuitively, the vast majority of these elements will be unconstrained. In traditional FD approaches, arrays are typically modeled using a list of variables, corresponding to *every* array element. With our transformation, we can drastically reduce the search space.

3 Background

The Theory of Arrays [Kroening and Strichman, 2016] is concerned with indexed data structures equipped with access and update operators. The *pure theory of arrays* is built on three sorts Arr, indices Ind, and elements $Elem$, and two operators:

$$select : Arr \times Ind \rightarrow Elem$$
$$store : Arr \times Ind \times Elem \rightarrow Arr$$

where $select(t, i)$ is the value at index i in array t, while $store(t, i, e)$ is an array identical to t, except in i, where the value e is now present. Formally, the theory is defined over the signature $\Sigma_A = \{select, store, =, \neq\}$, and has the following axioms :

$$\forall i, j.i = j \longrightarrow select(t, i) = select(t, j) \quad (1)$$
$$\forall i, j.i = j \longrightarrow select(store(t, i, e), i) = e \quad (2)$$
$$\forall i, j.i \neq j \longrightarrow select(store(t, i, e), j) = select(t, j) \quad (3)$$

Yet simple, this theory is hard to solve: the satisfiability problem for the conjunctive fragment is already NP-complete [Downey and Sethi, 1978]. The *extensional theory of arrays* is a common extension of the latter theory where (dis)equalities between entire arrays can be expressed. It is defined by adding the axiom of extensionality :

$$t = t' \longleftrightarrow \forall i, j.select(t, i) = select(t, j) \quad (4)$$

Finally, we encode *size constraints* by adding an operator $size : Arr \rightarrow \mathbb{N} \cup \{\infty\}$, satisfying the following axioms:

$$\forall t, t'.t = t' \longrightarrow size(t) = size(t') \quad (5)$$
$$\forall t, i, e.size(t) = size(store(t, i, e)) \quad (6)$$

In order to avoid dealing with undefined values, we only consider in the following formulas with the *well-defined access property*. A formula has the well-defined access property if it is guaranteed that array accesses will always occur within the array bounds. An arbitrary formula can be made to have this property by adding the *well-defined access condition* $1 \leq i \leq size(t)$ for every term $select(t, i)$ and $store(t, i, \_)$ it contains, in the vein of type correctness conditions from the PVS proof assistant [Owre et al., 1999].

Let ϕ be a formula in the theory of arrays. A model for ϕ, or ϕ-model, is a first order interpretation satisfying ϕ. ϕ is said to be *satisfiable* if it has a model, and is otherwise *unsatisfiable*.

CSPs. A valuation θ over a set of variables $Var \triangleq \{v_1, ..., v_n\}$, each associated to a domain $D_1, ..., D_n$ is a mapping from variables to values, such that, for each i, $\theta(v_i) \in D_i$. A constraint c over variables $vars(c)$ is defined by a set of valuations over $vars(c)$, these valuations being called solutions to c. A Constraint Satisfaction Problem (CSP) is a tuple $(Vars, Dom, C)$, where $Vars$ is a finite set of variables, Dom is a mapping from variables to finite domains, and C is a set of constraint, where each constraint is defined over a subset of $Vars$. A solution to a CSP $(Vars, Dom, C)$ is a valuation θ over $Vars$, such that, for each $v \in Vars$, $\theta(v) \in Dom(v)$, and for every constraint $c \in C$, $\theta_{/vars(c)}$ is a solution to c.

A CSP that has at least one solution is called *satisfiable*, otherwise it is called *unsatisfiable*. Deciding a CSP consists in determining if it is satisfiable or not.

Procedures for solving CSPs are called finite domain constraint solvers (FD in the rest of the paper). The key idea is local filtering: *propagators* remove from the variables' domains the values that cannot participate in any solution. Local filtering can lead to spectacular pruning of the search space, and has enabled FD solvers to tackle complex combinatorial problems, where brute force is not an option.

Encoding Array Problems as CSP. The usual approach to deal with array accesses in FD is by using the global constraint ELEMENT [Hentenryck and Carillon, 1988]. This constraint is well known, and numerous implementations are available [Carlsson et al., 1997; Tack et al., 2006; Nethercote et al., 2007]. For an index i, an element e, and an array A, modeled with n FD-variables representing every array element (thus only fixed size arrays can be encoded), ELEMENT(i, A, e) is true iff $A[i] = e$. Filtering algorithms for ELEMENT can usually ensure at least domain consistency over i and bound consistency over e and all the $A[k]$ [Carlsson et al., 1997], at a cost quadratic in n. In [Charreteur et al., 2009], another global constraint is proposed to deal with array updates, also with a propagation cost quadratic in the array size. By combining these global constraints with a congruence closure algorithm, the fdcc solver has been implemented and evaluated on randomly generated formula [Bardin and Gotlieb, 2012]. However, labeling-based search and local reasoning are ill-conditioned for large or unbounded arrays. That is why the transformation proposed in this paper is a necessary complement to any FD reasoning over arrays.

4 Formula Transformation

In this section we detail the process of obtaining a reduced form for a formula containing extended array constraints. Our transformation takes as input a quantifier-free formula ϕ containing fixed-size, bounded and/or unbounded arrays with access and update constraints, array (dis)equalities as well as other arbitrary FD-constraints on elements and indices. ϕ may contain disjunctions, and is assumed to be provided in negational normal form. *The transformation outputs an equisatisfiable formula ϕ^r containing only fixed-size arrays, amenable to efficient FD reasoning.*

4.1 Key Insights

The transformation takes advantage of the following key insights into the theory of arrays. *The first two insights are well-known in SMT (but not used in FD approaches), while the last two insights are at the core of our contribution.*

1. *Distinguished array cells.* Each array cell that is not referred to by an index expression in the formula does not impact the status of a given interpretation (model or not); i.e. two interpretations differing only on unreferenced array cells are equivalent. Hence, we can restrict our reasoning to referred array cells only, whose number is always finite and independent from array size (for quantifier-free formulas). *This is a first step toward handling unbounded arrays with finite reasoning.*

2. *Equality-based reasoning.* In pure array theory, only the (dis-)equality of index expressions are important – not their exact values. This is the basic enumeration strategy for solving pure array formulas (without domains nor size), requiring to enumerate 2^{N^2} different cases – but being absolutely independent from the domain of indices (N is the number of index expressions).

3. *Implicit equality encoding.* (new) With pure arrays, we can use an alternative method: bound every array to size N (the number of index expressions), and choose for indices values between $1..N$, in order to encode *in an implicit way* the dis-equalities of index expressions. The technique is correct and complete (this is proved in section 4.3), and shows two advantages over equality-based reasoning:

- finite-encoding: the formula contains now only fixed-size arrays, and we can solve it through standard FD approaches – even if the original formula was on unbounded arrays;
- efficiency: we can refine the technique in order to have a search space of $(N/ln(N))^N$, which is considerably smaller than for explicit equality-based reasoning.

4. *Proxying / array isolation.* (new) In the case of extended array constraints, we need an additional step of *proxying*, or *array isolation*, in order to keep the FD-reduction on arrays while allowing arbitrary constraints over index expressions. The idea is to separate the array-based reasoning on index expressions from other non-array reasoning on those index expressions, by introducing a proxy-variable $e^r \in 1..N$ for each index expression e, and ensuring overall formula consistency through a new dedicated (FD) constraint. Hence, arrays can be dealt with through the implicit encoding (the e^r)

while constraints over indexes are dealt with by the original problem variables.

4.2 Core Technique: Array Reduction

Our *array reduction* consists in four elementary steps, mostly involving rewriting or introducing new literals, and producing a series of equisatisfiable formulas.

Step 1: Preprocessing (Purification) The first step of the transformation is standard and consists in rewriting literals containing both array operators and arithmetic operators, such as $select(t, i+1) - x = y$ into conjunctions of literals that contain only one type of operator, introducing new variables where needed. For the previous literal, a *purified form* is $select(t, j) = e \wedge j = i+1 \wedge e - x = y$. Note that $=$ is to be understood as logical equality, and variables are assumed to be declared beforehand. *Purification* is a well known technique in SMT and is required in solver combination framework such as Nelson Oppen. It induces a linear growth of the input formula. We also flatten array constraints so as to make their arguments atomic. For example, $e = select(store(t, i, e), j)$ is rewritten as $t' = store(t, i, e) \wedge e = select(t', j)$, introducing one array variable t'. We call ϕ_{pure} the formula obtained after these operations. Finally, we rewrite array disequalities $t \neq t'$ as $select(t, j) \neq select(t', j)$, introducing a fresh variable j for every such disequality.

Step 2: Size Constraint Elimination We now proceed to remove every size constraints from ϕ_{array}, by encoding their semantics directly in the arithmetic part of the formula. Denote Arr the set of array atoms occurring in ϕ_{array}, and for every array $t \in Arr$ introduce a fresh variable s_t.

- replace every occurrence of $size(t)$ with s_t
- replace every array equality $t = t'$ with $t = t' \wedge s_t = s_{t'}$
- replace every array equality $t' = store(t, i, e)$ with $t' = store(t, i, e) \wedge s_t = s_{t'} \wedge i \in 1..s_t$
- replace every constraint $e = select(t, i)$ with $e = select(t, i) \wedge i \in 1..s_t$

We call ϕ_{elim} the formula obtained after eliminating size constraints from ϕ_{pure}.

Step 3: Index Reduction In ϕ_{elim}, we denote $Ind \triangleq \{i_1, ..., i_{N_i}\}$ the set of index variables, that is the variables that appear as the second argument in a $select$ or $store$ constraint. For each one of these variables, say i_k, we introduce a fresh variable i^r_k, and call $Ind^r \triangleq \{i^r_1, ..., i^r_{N_i}\}$ the reduced indices associated with Ind. The first step of index reduction consists in replacing every term of the form $select(t, i_k)$ in ϕ_{elim} with $select(t, i^r_k)$, and similarly every term $store(t, i_k, e)$ with $store(t, i^r_k, e)$. We emphasize that only the occurrences of indices as the second argument of an array constraint are replaced with their reduced counterparts. In particular, arithmetic constraints on indices remain unchanged. The idea is to isolate the arithmetic reasoning on indices, which is concerned about precise values, from the array reasoning, which is merely concerned about (dis)equalities.

Partially replacing indices with reduced indices amounts to under-constraining the problem, and as such, index reduction is not sound, that is, it may introduce solutions that do not satisfy the original formula. In order to ensure that the reduced indices are consistent with the original ones, we add the following consistency constraints:

$$i^r_k = i^r_l \Leftrightarrow i_k = i_l, \text{ for each } k < l$$

The number of consistency constraints is quadratic in N_i. Section 4.4 will show how to handle consistency efficiently in a FD solver using the new global constraint $consistent([i^r_1, ..., i^r_{N_i}], [i_1, ..., i_{N_i}])$.

As an additional step, we rename every array t in ϕ_{elim} as t^r (we refer to them as reduced arrays). This step is purely syntactic and optional, yet it helps avoiding confusion when we discuss the generation of ϕ-models from ϕ^r-models.

We call $\phi_{i.r.}$ the formula obtained as a result of index reduction on ϕ_{elim}.

Step 4: Size Fixing All arrays in $\phi_{i.r.}$ are unbounded, since size constraints were eliminated. Moreover, all array accesses occur on reduced indices, and the reduced indices appear nowhere except in array constraints and equality constraints (consistency constraints). For these reasons, it is possible to add the following fixed sizes for arrays and domains for reduced indices :

- add constraint $size(t^r) = N_i$ for every reduced array
- add domain constraints $i^r_k \in 1..\max_{l<k}(i^r_l) + 1$

The domain constraints presented here are sophisticated and allow for efficient solving of ϕ^r. Using the weaker domain $1..N_i$ (implied by the size constraints) for every reduced index also leads to a sound transformation. The stronger domains are obtained using the fact that the values of the reduced indices can be freely interchanged, as long as their arrangement with respect to equality is preserved. Technically speaking, the set of reduced indices admits all value symmetries. Using the stronger domain form amounts to statically breaking these symmetries.

4.3 Correctness

Array reduction enjoys the following theoretical properties :

Theorem 1 (Equisatisfiability)**.** *ϕ and ϕ^r are equisatisfiable.*

Sketch of Proof We focus on the equisatisfiability of ϕ_{elim} and ϕ^r (equisatisfiability of ϕ and ϕ_{pure} is well-documented, and ϕ_{pure} and ϕ_{elim} are equisatisfiable by the definition of size constraints). We first consider the case where ϕ_{elim} is a conjunction of literals, and let I be a ϕ_{elim}-model. For every non-array variable in $v \in vars(\phi_{elim})$, we define $I^r(v) \triangleq I(v)$. Since ϕ_{elim} and ϕ^r have the same arithmetic literals, we only need to define I^r for reduced arrays and indices, and check that it satisfies array literals, and consistency constraints. We define :

$$\begin{array}{lll} I^r(i^r_1) & \triangleq 1 & \\ I^r(i^r_k) & \triangleq I^r(i^r_l) & \text{if } \exists l < k, I(i_k) = I(i_l) \\ & \triangleq \max_{l<k}(I(i^r_l)) + 1 & \text{otherwise} \end{array}$$

It is easy to check, by induction, that $I^r(i^r_k)$s are well defined, and that this definition satisfies the consistency constraints as well

as the domain constraints for reduced indices. Now let t be an array variable in ϕ_{elim}. $I(t)$ is an integer sequence $(t_k)_{k>0}$ (all arrays in ϕ_{elim} are unbounded). We will define $I^r(t^r)$ as the finite sequence $(t^r_k)_{1\leq k\leq N_i}$ such that :

$$\begin{array}{rll} t^r_k & \triangleq t_{I(i_l)} & \text{if } \exists l, I^r(i^r_l) = k \\ & \triangleq 0 & \text{otherwise} \end{array}$$

It is again easy to check that $I^r(t^r)$ is well defined and satisfies the size constraint $size(t^r) = N_i$. It remains to show that I^r satisfies array literals in ϕ^r. This step is easy, remarking that for every index i and array t in ϕ, $t_{I(i)} = t^r_{I^r(i^r)}$, and unreferenced array elements all have value 0 in reduced array models, hence high level properties like array equalities are preserved. As a consequence, if ϕ_{elim} is satisfiable, then so is ϕ^r. The proof that ϕ_{elim} is satisfiable when ϕ^r is has the same structure (it is actually simpler since a ϕ^r already provides values for non-reduced indices). When ϕ_{elim} is not a conjunction of literals, the same proof applied to a satisfied clause in its disjunctive normal form shows that the result still holds. □

Theorem 2 (Model Extension)**.** *ϕ^r-models can be extended to ϕ-models, that is, models that agree on every variable common to ϕ and ϕ^r*

Sketch of Proof Let I^r be a ϕ^r-model. We obtain a ϕ_{elim}-model I_{elim} as follows :

- $I_{elim}(v) \triangleq I^r(v)$ for every non-reduced variable
- $I_{elim}(t) \triangleq (t_k)_{k>0}$ for array variables, where :

$$\begin{array}{rll} t_{I^r(i_k)} & \triangleq t^r_{I^r(i^r_k)} & \text{for each } k \\ & \triangleq 0 & \text{otherwise} \end{array}$$

The proof that I_{elim} is well defined, and actually a model for ϕ_{elim} is similar to the previous proof. Remarking that $vars(\phi) \subset vars(\phi_{elim})$, a ϕ-model I is defined as follows :

- $I(v) \triangleq I_{elim}(v)$, for every non array variable $v \in vars(\phi)$
- $I(t) \triangleq t_{(1\leq k\leq I_{elim}(s_t))}$, for array variables

The proof that I is a ϕ-model is routine. □

Theorem 3 (Finite Reasoning)**.** *ϕ^r can be solved using FD techniques, especially consistency constraints.*

Proof Arithmetic literals come with finite domains. FD solving for arithmetic is well documented. Arrays and indices in ϕ^r all have fixed size and finite domains (introduced at transformation time), hence array accesses and updates can be modeled using existing FD techniques. Consistency constraints are finite in number, and all have the form $i^r = j^r \Leftrightarrow i = j$. This constraint can be encoded in most FD frameworks using reified constraints. Section 4.4 shows a more efficient handling. □

4.4 Efficiency

We discuss how to efficiently maintain consistency while solving ϕ^r. For that goal we introduce the global *consistent* constraint. This constraint takes as input two lists of variables having the same length n. Two list of values $[u_1, ..., u_n]$and $[v_1, ..., v_n]$ satisfy the *consistent* constraint when they have the same arrangement with respect to equality, that is, when $u_i = u_j$, if, and only if, $v_i = v_j$, for every i and j in $1..n$.

We now show how to propagate the $consistent(L_1, L_2)$ constraint. The constraint internally maintains two lists I_1 and I_2 of integer values i such that $L_1[i]$ (respectively $L_2[i]$) is assigned. The constraint is woken whenever a variable in L_1 or L_2 becomes assigned, and uses the following propagation algorithm :

```
woken on L1[i] ← v (alternatively L2[i] ← v);
add i to I1 (alternatively I2);
for k, l in I1 do
    if L1[k] = L1[l] then
        Dom(L2[k]) ← Dom(L2[k]) ∩ Dom(L2[l]);
        Dom(L2[l]) ← Dom(L2[k]) ∩ Dom(L2[l]);
    else
        if k is in I2 then
            Dom(L2[l]) ← Dom(L2[l])\{L2[k]};
        end
        if l is in I2 then
            Dom(L2[k]) ← Dom(L2[k])\{L2[l]};
        end
    end
end
similar loop for k, l in I2
```

Algorithm 1: Propagator for $consistent(L_1, L_2)$

Using a global consistency constraint is more efficient than using reified constraints in our applications. A search strategy that worked well in practice is to label reduced indices first, since their domains are generally smaller than other variables in ϕ^r and instantiating these variables leads to strong propagation, both in array and consistency constraints.

When using a consistency constraint, the overall size blowup of ϕ^r relative to ϕ is only linear.

Theorem 4 (Solution Space Reduction)**.** *Every ϕ-model can be obtained as an extension of a ϕ_r-model. Hence, the solution space for ϕ_r is a reduced version of that of ϕ, all solutions are preserved but irrelevant indices are abstracted.*

Sketch of Proof The ϕ^r-model is constructed using the definitions in the proof of Theorem 1, with minor adaptations. □

Theorem 5 (Search Space Reduction)**.** *Let Ind be the set of index expressions in ϕ and Ind^r the set of reduced index in ϕ^r. Two ϕ^r-models not agreeing on Ind^r extend to ϕ-models not agreeing on Ind, while the converse does not hold in general.*

Sketch of Proof Let $Ind^r \triangleq \{i^r_1, ..., i^r_n\}$, and call I and I' two ϕ^r-models such that $I(i^r_k)$ and $I'(i^r_k)$ differ, for some k. Because of the domains of reduced indices, we know that, $I(i^r_1) = I'(i^r_1) = 1$, and $I(i^r_2)$, as well as $I'(i^r_2)$, is either 1 or 2. If they differ, then one of these models gives the same value to i^r_1 and i^r_2, while the other does not. With a similar argument we prove by induction that there will always be two indices i^r_l and i^r_m such that $I(i^r_l) = I(i^r_m)$ and $I'(i^r_l) \neq I'(i^r_l)$, exchanging the roles of I and I' if needed. The result then follows from the fact that I and I' satisfy the consistency constraints. □

Search Space Comparison We consider a pure array formula ϕ with unbounded arrays. That is, ϕ^r has no arithmetic literal. In this case, deciding the formula only requires

enumerating the valuations for Ind^r the reduced indices in ϕ^r. This is because array elements appear only at most in (dis)equality constraints, so finding values for these elements from a non-conflicting index valuation is easy. As hinted by the previous proof, different valuations on Ind^r correspond to different arrangements of Ind^r with respect to equality. In fact, one can show that valuations on Ind^r correspond exactly to arrangements of Ind^r with respect to equality (or in other terms, to equivalence relations over Ind^r). Hence there are B_n such valuations, where B_n is the n-th Bell number, and n is the size of Ind^r. As a comparison, many SMT approaches rely on introducing case splits on the (dis)equality of i and j for every read-over-write term $select(store(t, i, e), j)$. This approach introduces 2^{n^2} case splits in the worst case.

It is known that $B_n = \mathcal{O}\left(\left(\frac{n}{\log n}\right)^n\right)$ [Berend and Tassa, 2010], while $2^{n^2} = (2^n)^n$. Although this comparison does not take into account the various optimisations and heuristics used in practice by SMT solvers and FD solvers, the difference is significant. For small values of n, the difference in search space size is already huge, for example $B_5 = 52$ while $2^{5^2} \geq 33.10^6$.

5 Experimental Evaluation

Implementation Our implementation is built on top of fdcc [Bardin and Gotlieb, 2012]. fdcc relies on update constraints and SICStus clpfd for arithmetic constraints. The implementation is approximately 2000 lines of Prolog, and includes an interface to the SMT-LIB [Barrett et al., 2015] format (only array and integer related theories are supported).

Goal and Protocol The experiments presented below aims at evaluating the benefits of array reduction. The goal is to answer precisely the following questions:

1. Does array reduction indeed lift standard FD techniques to unbounded formulas?
2. How useful is array reduction when implemented on top of FD solvers for solving formulas with bounded arrays?
3. How does array-augmented FD solvers perform w.r.t. top-class SMT solvers, especially do we manage to bridge the gap between FD and SMT?

We consider a benchmark of 2200 formulas obtained as follow. First we take the 550 array formulas from the SMT-COMP benchmark, the standard competition in SMT solving – with formulas coming essentially from hard verification-oriented industrial and academic case-studies. We consider here only pure array formulas (no additional arithmetic constraints). They typically include up to a hundred variables and dozens of array constraints (including long store chains), with the largest ones containing more than one hundred arrays, 60 distinct indices and a thousand constraints (including 120 array updates). Most formulas include extensionality constraints, but no size constraint. Then, we automatically duplicate these formulas with additional size constraints of 10, 100, 1000. These sizes, and small ones in particular, are representative of the ones found in real programs.

In our experiments, we compared fd which is the standard clpfd library of SICStus Prolog augmented with an implementation of the store operator using the global constraint interface [Charreteur et al., 2009], fdcc [Bardin and Gotlieb, 2012] which augments clpfd with congruence closure reasoning over arrays, fd^r and fdcc^r which are similar to fd and fdcc but augmented with array reduction. The augmented features for these four tools are fully implemented in Prolog on top of clpfd. We could not use MiniZinc [Nethercote et al., 2007] and Gecode [Schulte and Tack, 2005] as these solvers do not provide any constraint for handling the store operator which is present in all the formulas. For the sake of completeness, we also ran four SMT-solvers which are among the best competitors of the SMT competition: Yices [Dutertre, 2014], MathSAT [Cimatti et al., 2013], CVC4 [Barrett et al., 2011] and the Microsoft Z3 solver [De Moura and Bjørner, 2008] [1]. *These SMT-solvers are among the best-known approaches for the theory of arrays, they result from more than 10 years of intensive development by teams of experienced engineers and are finely tuned for SMTCOMP.*

We compare each solver on the number of successfully resolved formulas (solver answers were checked against each other and the formula oracle, no conflict was reported), with a timeout set to 30 seconds – this is a rather low value, yet it is representative of timeouts used in some verification settings, where thousands of constraints must be solved. The results for a timeout of 120 seconds are also shown without discussion, since there is only very little difference. Experiments were run on a Intel(R) Core(TM) i7-5600U (2,6 GHz) (2 cores), 16GB RAM, running Linux Fedora 22.

Results and Conclusion Results are presented in Table 2.

First, it can be seen that array reduction does allow FD techniques *to solve unbounded formulas in practice*, and actually it allows to solve a large majority of the formulas (526/550 for fd^r and fdcc^r).

Second, array reduction allows a *dramatic improvement* of standard FD techniques on *fixed-size arrays* (between 2.5x and 6.5x more formulas), and the larger size, the larger improvement (for size 1000, fd^r solves 526 formulas, while fd alone solves only 79 formulas); interestingly, the reduction allows also to bridge the gap between fd and fdcc. For small array sizes. fd-like cheaper propagation even gives fd^r a slight advantage (544 vs 536) since it can exhaust all valuations for reduced indices more quickly. Yet, array reduction does not amount to adding full symbolic reasoning to FD solvers, and there are classes of formulas (not represented in SMT-COMP) that require fdcc's global reasoning to be solved.

Finally, array reduction allows *FD to compete with the best SMT approaches*: our technique clearly surpasses SMT on small size arrays – at worst 536 vs at best 463 (small sizes yield many pigeonhole-like problems, notoriously hard to solve with SMT), while it is only slightly inferior on larger-size and unbounded formulas – 526 vs 550. Arrays of small

[1] Yices, MathSAT, Z3 and CVC4, in this order, achieved the four first places in the 2016 SMT-COMP.

Table 2: Comparison (FD, SMT solvers, and our approach)

size	#f	fd	fdcc	fd^r	$fdcc^r$	cvc4	z3	math sat	yices
Timeout 30 s									
10	550	212	222	**544**	**536**	**451**	**463**	**376**	**376**
10^2	550	123	137	526	526	538	550	550	550
10^3	550	**79**	**92**	**526**	**526**	538	550	550	550
∞	550	xxx	xxx	526	526	547	550	550	550
Timeout 120 s									
10	550	212	222	544	540	460	463	376	376
10^2	550	123	137	540	526	547	550	550	550
10^3	550	99	113	526	526	547	550	550	550
∞	550	xxx	xxx	526	526	548	550	550	550

. fd^r and fd: Constraint solver, with and without array reduction
. $fdcc^r$ and fdcc: Hybrid solver, with and without array reduction
. size: array size – #f: n. of formulas

sizes are ubiquitous in real-world programs, hence they are of particular importance in practice.

6 Related Work

Standard SMT and FD approaches for arrays and how they relate to our reduction-based method have already been presented and discussed through the paper (especially, see Table 1, Sections 3 and 4.1).

FD While array accesses have been dealt with for a long time through constraint ELEMENT [Hentenryck and Carillon, 1988], only very few work consider array updates [Gotlieb, 2009]. For example, both Minizinc [Nethercote et al., 2007] and Gecode [Schulte and Tack, 2005] propose the former but not the latter. All these approaches are size-dependent and cannot deal with unbounded arrays. While string constraints [Jaffar, 1990] are highly expressive, size constraints are often weak or not well treated. Indeed, solving constraints over regular expressions extended with arithmetic is often undecidable. Depending on the approach, sizes must be bounded, sizes must be unbounded, or the decision procedure does not guarantee termination. The fdcc approach [Bardin and Gotlieb, 2012] complements the standard local filtering-based FD reasoning on array with global symbolic reasoning to produce an efficient solver for *fixed-size arrays*. Array reduction and fdcc are complementary. Finally, array reduction is not about automatically finding symmetries in a given CSP. Rather, we take advantage of existing symmetries for reducing the initial problem to an efficient finite-domain problem.

SMT Standard symbolic approaches for pure arrays complement symbolic read-over-write preprocessing (in the vein of fdcc) with enumeration on (dis-)equalities, yielding a potentially huge search space.

New array lemmas can be added on-demand or incrementally discovered through an abstraction-refinement scheme [Brummayer and Biere, 2009]. Size and arithmetic constraints can in principle be recovered through the Nelson-Oppen solver combination scheme [Nelson and Oppen, 1979], but the communication cost can be much more expensive than satisfiability checking [Bruttomesso et al., 2009] on *non-convex theories* – such as array theory, as it requires to propagate all implied disjunctions of equalities. Delayed theory cooperation [Bozzano et al., 2005; Bruttomesso et al., 2009] requires only equality propagation, at the price of adding new Boolean variables for all potential variable equalities. Model-based theory cooperation [Marre and Blanc, 2005] aims at mitigating this overhead through lazy equality propagation. Decision procedures have been developed for expressive extensions of the array theory, such as the array property fragment [Bradley et al., 2006], which enables limited forms of quantification over indices, and arithmetic constraints. While such theories are more expressive than our extended array constraints, the associated decision procedures are highly expensive (translation to Presburger arithmetic). Moreover, these techniques require indices to be integers (bitvectors, for example, are not supported), while our approach is mostly independent of the elements' and indices' types (only equality and disequality are required over elements, while indices additionally require an ordering).

7 Conclusion

This papers introduces a formula transformation that can produce a fixed-size array formula equisatisfiable to any formula with extended array constraints, including extensional unbounded arrays, and combination with arithmetic constraints on indices and elements. In addition, the transformation induces a powerful search space reduction and has remarkable properties including a strong correspondence between the models to the input and transformed formulas. This work opens the way for automated constraint-based reasoning on large classes of data structures, with reasonable theory combination costs, and follows the trend initiated in previous works of proposing CP as a viable alternative to the well established techniques in automated proof, relying largely on the use of DPLL based SMT solvers.

References

[Bardin and Gotlieb, 2012] Sébastien Bardin and Arnaud Gotlieb (2012). Fdcc: A combined approach for solving constraints over finite domains and arrays. In *Proceedings of the 9th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR'12, pages 17–33, Berlin, Heidelberg. Springer-Verlag.

[Bardin and Herrmann, 2008] Sébastien Bardin and Philippe Herrmann (2008). Structural testing of executables. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 22–31.

[Bardin et al., 2010] Sébastien Bardin, Philippe Herrmann, and Florian Perroud (2010). An alternative to sat-based approaches for bit-vectors. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 84–98. Springer.

[Barrett et al., 2011] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli (2011). CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177.

[Barrett et al., 2015] Clark Barrett, Pascal Fontaine, and Cesare Tinelli (2015). The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[Berend and Tassa, 2010] Daniel Berend and Tamir Tassa (2010). Improved bounds on bell numbers and on moments of sums of random variables. *Probability and Mathematical Statistics*, 30(2):185–205.

[Botella et al., 2006] Bernard Botella, Arnaud Gotlieb, and Claude Michel (2006). Symbolic execution of floating-point computations: Research articles. *Softw. Test. Verif. Reliab.*, 16(2):97–121.

[Bozzano et al., 2005] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Silvio Ranise, Peter van Rossum, and Roberto Sebastiani (2005). Efficient satisfiability modulo theories via delayed theory combination. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, pages 335–349.

[Bradley et al., 2006] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma (2006). What's decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'06, pages 427–442, Berlin, Heidelberg. Springer-Verlag.

[Brummayer and Biere, 2009] Robert Brummayer and Armin Biere (2009). Lemmas on demand for the extensional theory of arrays. *JSAT*, 6(1-3):165–201.

[Bruttomesso et al., 2009] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani (2009). Delayed theory combination vs. nelson-oppen for satisfiability modulo theories: a comparative analysis. *Ann. Math. Artif. Intell.*, 55(1-2):63–99.

[Carlsson et al., 1997] Mats Carlsson, Greger Ottosson, and Björn Carlson (1997). An open-ended finite domain constraint solver. In *Proceedings of the9th International Symposium on Programming Languages: Implementations, Logics, and Programs: Including a Special Trach on Declarative Programming Languages in Education*, PLILP '97, pages 191–206, London, UK, UK. Springer-Verlag.

[Charreteur et al., 2009] Florence Charreteur, Bernard Botella, and Arnaud Gotlieb (2009). Modelling Dynamic Memory Management in Constraint-Based Testing. *Journal of Systems and Software*, 82(11):1755–1766.

[Cimatti et al., 2013] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani (2013). The MathSAT5 SMT Solver. In Nir Piterman and Scott Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer.

[De Moura and Bjørner, 2008] Leonardo De Moura and Nikolaj Bjørner (2008). 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, pages 337–340, Berlin, Heidelberg. Springer-Verlag.

[Downey and Sethi, 1978] Peter J. Downey and Ravi Sethi (1978). Assignment commands with array references. *J. ACM*, 25(4):652–666.

[Dutertre, 2014] Bruno Dutertre (2014). Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer.

[Gotlieb, 2009] Arnaud Gotlieb (2009). Euclide: A constraint-based testing framework for critical c programs. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 151–160. IEEE.

[Gotlieb et al., 2010] Arnaud Gotlieb, Michel Leconte, and Bruno Marre (2010). Constraint solving on modular integers. In *ModRef Worksop, associated to CP'2010*, Saint-Andrews, United Kingdom.

[Hentenryck and Carillon, 1988] Pascal Van Hentenryck and Jean-Philippe Carillon (1988). Generality versus specificity: An experience with AI and OR techniques. In *Proceedings of the 7th National Conference on Artificial Intelligence. St. Paul, MN, August 21-26, 1988.*, pages 660–664.

[Jaffar, 1990] Joxan Jaffar (1990). Minimal and complete word unification. *J. ACM*, 37(1):47–85.

[Kroening and Strichman, 2016] Daniel Kroening and Ofer Strichman (2016). *Decision Procedures: An Algorithmic Point of View (Chapter Arrays)*, pages 157–172. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Marre and Blanc, 2005] Bruno Marre and Benjamin Blanc (2005). Test selection strategies for lustre descriptions in gatel. *Electron. Notes Theor. Comput. Sci.*, 111:93–111.

[Nelson and Oppen, 1979] Greg Nelson and Derek C. Oppen (1979). Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257.

[Nethercote et al., 2007] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack (2007). Minizinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, pages 529–543.

[Owre et al., 1999] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert (1999). *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA.

[Schulte and Tack, 2005] Christian Schulte and Guido Tack (2005). Views and iterators for generic constraint implementations. In *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, pages 817–821.

[Tack et al., 2006] Guido Tack, Christian Schulte, and Gert Smolka (2006). Generating propagators for finite set constraints. In *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, pages 575–589.