FDCC: A combined approach for solving constraints over Finite Domains and Arrays *

Sébastien Bardin[1] and Arnaud Gotlieb[2,3]

[1] CEA, LIST, Gif-sur-Yvette, F-91191, France
sebastien.bardin@cea.fr
[2] INRIA Rennes Bretagne Atlantique, Rennes, France
arnaud.gotlieb@inria.fr
[3] Certus V&V Center, Simula Research Lab, Oslo, Norway

Abstract. Arrays are ubiquitous in the context of software verification. However, effective reasoning over arrays is still rare in CP, as local reasoning is dramatically ill-conditioned for constraints over arrays. In this paper, we propose an approach combining both global symbolic reasoning and local filtering in order to solve constraint systems involving arrays (with accesses, updates and size constraints) and finite-domain constraints over their elements and indexes. Our approach, named FDCC, is based on a combination of a congruence closure algorithm for the standard theory of arrays and a CP solver over finite domains. The tricky part of the work lies in the bi-directional communication mechanism between both solvers. We identify the significant information to share, and design ways to master the communication overhead. Experiments on random instances show that FDCC solves more formulas than any portfolio combination of the two solvers taken in isolation, while overhead is kept reasonable.

1 Introduction

Context. Constraint resolution is an emerging trend in software verification [25], either to automatically generate test inputs or formally prove some properties of a program. Program analysis involves solving so-called Verification Conditions (VCs), i.e. checking the satisfiability of a formula either by providing a solution (*sat*) or showing there is none (*unsat*). While most techniques are based on SMT (Satisfiability Modulo Theory), a few verification tools [3, 10, 15, 20] rely on Constraint Programming over Finite Domains, denoted CP(FD). CP(FD) is appealing here because it allows to reason about some fundamental aspects of programs notoriously difficult to handle, like floating-point numbers [6], bounded non-linear integer arithmetic, modular arithmetic [16] or bitvectors [4]. Some experimental evaluations [4, 11] suggest that CP(FD) could be an interesting alternative to SMT for certain classes of VCs.

* Work partially funded by ANR (grants ANR-08-SEGI-006).

The problem. Yet the effective use of CP(FD) in program verification is limited by the absence of effective methods to handle complex constraints over arrays. While array accesses are handled for a long time through the ELEMENT constraint [17], array updates have been dealt with only recently [10], and in both cases the reasoning relies only on local (consistency-based) filtering. This is insufficient to handle constraints involving long chains of accesses and updates arising in program verification.

On the other hand, the theory of array is well-known in theorem proving [8]. The standard theory of array considered there cannot express size constraints over arrays or finite-domain constraints over elements and indexes. One must use a combination of two decision procedures, one for the array part and one for the index / element part, through a standard cooperation framework like the Nelson-Oppen (NO) scheme [22]. Unfortunately, finite-domain constraints cannot be integrated into NO (eligible theories must have an infinite model [23]).

Contributions. This paper addresses the problem of designing an efficient CP(FD) approach for solving conjunctive quantifier-free formulas combining arrays with size constraints and finite-domain constraints over indexes and elements. Our main guidelines are (1) to combine global symbolic deduction mechanisms with local filtering in order to achieve better deductive power than both technique taken in isolation, (2) to keep communication overhead as low as possible, while going beyond a purely portfolio combination of the two approaches, (3) to design a combination scheme allowing to re-use any existing FD solver in a black box manner, with minimal and easy-to-implement API .

Our main contributions are the following:

1. We design FDCC, an original decision procedure built upon a (new) lightweight congruence closure algorithm for the theory of arrays, called CC in the paper, interacting with a (standard) filtering-based CP(FD) solver, called FD. To the best of our knowledge, it is the first collaboration scheme including a finite-domain CP solver and a Congruence Closure solver for array constraint systems. Moreover, the combination scheme, while more intrusive than NO, is still high-level. Especially, FD can be used in a black-box manner through a minimal API, and large parts of CC are standard.
2. We bring new ideas to make both solvers cooperate through bi-directional constraint exchanges and synchronisations. We identify important classes of information to be exchanged, and propose ways of doing it efficiently : on the one side, the congruence closure algorithm can send equalities, disequalities and ALLDIFFERENT constraints to FD, while on the other side, FD can deduce new equalities / disequalities from local filtering and send them to CC. In order to master the communication overhead, a *supervisor* queries explicitly the most expensive computations, while cheaper deductions are propagated asynchronously.
3. We propose an implementation of our approach written on top of SICStus `clpfd`. Through experimental results on random instances, we show that FDCC systematically solve more formulas that CC and FD taken in isolation. FDCC performs even better than the best possible portfolio combination of

the two solvers. Moreover, FDCC shows only a reasonable overhead over CC and FD.

2 Motivating examples

Prog1
```
int T[100]; ...
int e=T[i]; int f=T[j];
if (e != f && i = j) { ...
```

Prog2
```
int T[2]; ...
int e=T[i]; int f=T[j]; int g=T[k];
if (e != f && e != g && f != g) { ...
```

Fig. 1. Programs with arrays

We use the two programs of Fig. 1 as running examples. First, consider the problem of generating a test input satisfying the decision in program Prog1 of Fig. 1. This involves solving a constraint system with array accesses, namely

$$\text{ELEMENT}(i, T, e), \text{ELEMENT}(j, T, f), e \neq f, i = j \tag{1}$$

where T is an array of variables of size 100, and $\text{ELEMENT}(i, T, e)$ means $T[i] = e$. A model of this constraint system written in COMET [21] did not provide us with an *unsat* answer within 60 minutes of CPU time on a standard machine. In fact, as only local consistencies are used in the underlying solver, the system cannot infer that $i \neq j$ is implied by the three first constraints. On the contrary, a SMT solver such as Z3 [13] immediately gives the expected result, using a global symbolic decision procedure for the standard theory of arrays.

Second, consider the problem of producing a test input satisfying the decision in program Prog2 of Fig. 1. It requires solving the following constraint system:

$$\text{ELEMENT}(i, T, e), \text{ELEMENT}(j, T, f), \text{ELEMENT}(k, T, g), e \neq f, e \neq g, f \neq g \tag{2}$$

where T is an array of size 2. A symbolic decision procedure for the standard theory of arrays returns (wrongly) a *sat* answer here (size constraints are ignored), while the formula is unsatisfiable since $T[i], T[j]$ and $T[k]$ cannot take three distinct values. A symbolic approach for arrays must be combined either with an explicit encoding of all possible values of indexes, or with the theory of integer linear arithmetic via NO. However, both solutions are expensive, the explicit encoding of domains adds many disjunctions (requiring enumeration at the SAT solver level), and combination of arrays and integers requires to find all implied disjunctions of equalities. On this example, a CP solver over finite domains can also fail to return *unsat* in a reasonable amount of time if it starts labelling on elements instead of indexes, as nothing prevents to consider constraint stores where $i = j$ or $i = k$ or $j = k$: there is no *global* reasoning over arrays able to deduce from $T[i] \neq T[j]$ that $i \neq j$.

3 Background

We describe hereafter the standard theory of arrays, existing CP(FD) constraints over arrays and the congruence closure algorithm. In the following, logical theories are supposed to be quantifier-free. Moreover, we are interested in conjunctive fragments.

The theory of arrays. The theory of arrays has signature $\Sigma_A = \{select, store, =, \neq\}$, where $select(T, i)$ returns the value of array T at index i and $store(T, i, e)$ returns the array obtained from T by putting element e at index i, all other elements remaining unchanged. The theory of arrays is typically described using the *read-over-write semantics*. Besides the standard axioms of equality, three axioms dedicated to *select* and *store* are considered. Axiom (3) is an instance of the classical *functional consistency* axiom (FC), while (4) and (5) are two variations of the *read-over-write* principle (RoW).

$$i = j \longrightarrow select(T, i) = select(T, j) \tag{3}$$

$$i = j \longrightarrow select(store(T, i, e), j) = e \tag{4}$$

$$i \neq j \longrightarrow select(store(T, i, e), j) = select(T, j) \tag{5}$$

The theory of arrays is difficult to solve: the satisfiability problem for its *conjunctive fragment* is already NP-complete [14].

The theory of arrays by itself does not express anything about the size of arrays or the domains of indexes and elements. Moreover, the theory presented here is *non-extensional*, meaning that it can reason on array elements but not on arrays themselves. For example, $A[i] \neq B[j]$ is permitted, while $A \neq B$ and $store(A, i, e) = store(B, j, v)$ are not.

CP(FD) and arrays. In CP(FD) solvers, *select* constraints over arrays are typically handled with constraint ELEMENT (i, A, v) [17]. The constraint holds iff $A[i] = v$, where i, v are finite domain variables and A is a fixed-size sequence (array) of constants or finite domain variables. Local filtering algorithms are available for ELEMENT at quadratic cost [7]. Interestingly, ELEMENT can reason on array size by constraining the domain of indexes. Filtering algorithms for *store* constraints over arrays have been defined in [10], with applications to software testing. Aside dedicated propagators, *store* could also be removed through the introduction of reified case-splits following axioms (4) and (5), but this is notoriously inefficient in CP(FD).

Terminology. In this paper, we consider filtering over ELEMENT as implementing *local reasoning*, while *global reasoning* refers to deduction mechanisms working on a global view of the constraint system, e.g. taking into account all *select*/*store*.

The congruence closure algorithm. Computing the congruence closure of a relation over a set of terms has been studied by Nelson and Oppen [23]. The algorithm uses a *union-find structure* to represent the equivalence relation between terms as its quotient set, i.e., the set of all equivalence classes. Basically, each

class of equivalence has a unique witness and each term is (indirectly) linked to its witness. Adding an equality between two terms amounts to choose one term's witness to be the witness of the other term. Disequalities inside the same equivalence class lead to *unsat*, otherwise the formula is *sat*. Smart handling of "witness chains" ensures very efficient implementations. Congruence closure is different from Prolog unification in that it allows to deal with non-free algebra, for example if we want to express that $f(a) = g(b) = 3$.

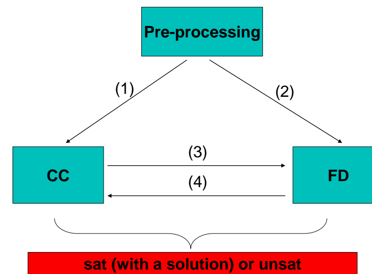
Remark 1. In the (standard) congruence closure algorithm, all implied equalities are saturated (made explicit), while disequalities deduced from the FC axiom are left implicit: adding the corresponding equality will lead to an *unsat* answer, but it is not easy to retrieve all these inequalities.

4 Combining CC and FD

4.1 Overview

Our approach is based on combining symbolic global reasoning for arrays and local filtering resolution. The framework, sketched in Fig. 2, is built over three main ingredients:

1. local filtering algorithms for arrays and other constraints on elements and indexes (called FD),
2. a lightweight global symbolic reasoning over array accesses and updates (called CC),
3. a new bi-directional communication mechanism between the two decision procedures above.



(1) subformula with accesses, updates, $=$, $\neq$
(2) whole initial formula
(3) implied $=$ and $\neq$, cliques of disequalities
(4) implied $=$ and $\neq$ (through filtering)

Fig. 2. An overview of FDCC

Let φ be a conjunction of equalities, disequalities, array accesses (*select*) and updates (*store*), constraint on the size of arrays and other (arbitrary) constraints over elements and indexes. Our procedure takes φ as input, and returns a verdict that can be either *sat* or *unsat*. First, the formula φ is pre-processed and dispatched between CC and FD. More precisely, equalities and disequalities as well as array accesses and updates go to both solvers. Constraints over elements and indexes go only to FD. The two solvers exchange the following information: CC can communicate new equalities and disequalities among variables to FD, as well as sets of variables being all different

(i.e., cliques of disequalities); FD can also communicate new equalities and disequalities to CC, based on domain analysis of variables. The communication mechanism and the decision procedures are described more precisely in the rest of this section.

4.2 The CC decision procedure

We can adapt the standard congruence closure algorithm into a semi-decision procedure CC for arrays. By semi-decision procedure, we mean here that all deductions made by the procedure are correct w.r.t. array axioms, but these deductions may not be sufficient to conclude to *sat* or *unsat*. CC is correct (verdict can be trusted) but not complete (may output "maybe").

For the sake of clarity we refine the set of array axioms given in Section 3 into an equivalent set of five more operational axioms:

(FC-1) $i = j \longrightarrow select(T, i) = select(T, j)$

(FC-2) $select(T, i) \neq select(T, j) \longrightarrow i \neq j$

(RoW-1) $i = j \longrightarrow select(store(T, i, e), j) = e$

(RoW-2) $i \neq j \longrightarrow select(store(T, i, e), j) = select(T, j)$

(RoW-3) $select(store(T, i, e), j) \neq e \longrightarrow i \neq j$

The congruence closure algorithm is adapted in the following way to handle these five different rules. Functional consistency rules FC-1 and FC-2 are standardly handled with slight extension of congruence closure [23]. To cope with RoW-1 and RoW-3, we *close* the set of constraints in CC by adding the equality $select(store(T, i, e), i) = e$ for each term $store(T, i, e)$, then RoW-1 and RoW-3 become specific instances of FC-1 and FC-2. Finally, for RoW-2 we add a mechanism of *delayed evaluation* inside CC: for each term $select(store(T, i, e), j)$, we put (T, i, e, j) in a watch list, and when $i \neq j$ is proved, we deduce the equality $select(store(T, i, e), j) = select(T, j)$.

Note that while implied disequalities are left implicit in the congruence closure procedure, in CC we close the set of disequalities (especially through FC-2) in order to benefit as much as possible from rules RoW-2 and RoW-3.

Obviously this polynomial-time procedure is not complete (recall that the problem is NP-complete), however we think that it is a nice trade-off between standard congruence closure (no array axiom taken into account) and full closure (exponential cost because of the introduction of case-splits for RoW-* rules).

4.3 The FD decision procedure

We use existing propagators and domains for constraints over finite domains. Our approach requires at least array constraints for *select/store* operations, and support of ALLDIFFERENT constraint [24] is a plus. Array constraints can be implemented either with the standard ELEMENT constraint and reified disjunctions, or (more efficiently) with the *load_element* and *store_element* constraints [10].

4.4 Cooperation between CC and FD

The cooperation mechanism involves both to know which kind of information can be exchanged, and how the two solvers synchronise together. Our main contribution here is twofold: we identify interesting information to share, and we design a method to tame the communication cost.

Communication from CC to FD. Our implementation of CC maintains the set of disequalities and therefore both equalities and disequalities can be easily transmitted to FD. Interestingly, maintaining disequalities allows to communicate also ALLDIFFERENT constraints. More precisely, any set of disequalities can be represented by an undirected graph where each node corresponds to a term, and there is an edge between two nodes iff there is a disequality between the corresponding terms. Finding the cliques[4] of the graph permits one to identify ALLDIFFERENT constraints that can be transmitted to FD. These cliques can be sought dynamically during the execution of the congruence closure algorithm. Since finding *a largest clique* of a graph is NP-complete, restrictions have to be considered. Practical choices are described in Sec. 5.1.

Communication from FD to CC. FD may discover new disequalities and equalities through filtering. For example, consider the constraint $z \geq x \times y$ with domains $x \in 3..4$, $y \in 1..2$ and $z \in 5..6$. While no more filtering can be performed, we can still deduce that formulas $x \neq y$, $x \neq z$ and $y \neq z$ hold, and transmit them to CC. Yet, this information is left implicit in the constraint store of FD and need to be checked explicitly. But there is a quadratic number of pairs of variables, and (dis-)equalities could appear at each filtering step. Hence, the eager generation of all domain-based (dis-)equalities must be temperated in order to avoid a combinatorial explosion. We propose efficient ways of doing it hereafter.

Synchronisation mechanisms: how to tame communication costs. A purely asynchronous cooperation mechanism with systematic exchange of information between FD and CC (through suspended constraints and awakening over domain modification), as exemplified in Fig. 2, appeared to be too expensive in practise. We manage this problem through a reduction of the number of pairs of variables to consider (**critical pairs**, see after) and a **communication policy** allowing tight control over expensive communications.

1. We use the following **communication policy**:

- cheap communications are made in an asynchronous manner;
- expensive communications, on the other hand, are made only on request, initiated by a *supervisor*;
- the two solvers run asynchronously, taking messages from the supervisor;
- the supervisor is responsible to dispatch formulas to the solvers, to ensure a consistent view of the problem between FD and CC, to forward answers of one solver to the other and to send queries for expensive computations.

[4] A clique is a subset of the vertices such that every two vertices in the subset are connected by an edge.

It turns out that all communications from CC to FD are cheap, while communications from FD to CC are expensive. Hence, it is those communications which are made only upon request. Typically, it is up to the supervisor to explicitly ask if a given pair of variables is equal or different in FD. Hence we have a total control on this mechanism.

2. We also reduce the number of pairs of variables to be checked for (dis-)equality in FD, by focusing only on pairs whose disequality will surely lead to new deductions in CC (i.e., pairs involved in the left-hand side of rules *FC-2*, *RoW-2* and *RoW-3*). Such pairs of variables are said to be *critical.* Considering the five deduction rules of Section 4.2, the **set of all critical pairs** is defined by:

- for each array T, all pairs $(select(T, i), select(T, j))$,
- for each term v of the form $select(store(T, i, e), j)$, pairs (i, j) and (e, v).

Yet, the number of such pairs is still quadratic, not in the number of variables but in the number of *select*. We choose to focus our attention only on the second class of critical pairs: they capture the specific essence of array axioms (besides FC) and their number is only *linear* in the number of *select*.

In practise, it appears that this reduction is manageable while still bringing interesting deductive power.

Complete architecture of the approach. A detailed architecture of our approach can be found in Fig. 3. Interestingly, CC and FD do not behave in a symmetric way: CC transmits systematically to the supervisor all new deductions made and cannot be queried, while FD transmits equalities and disequalities only upon request from the supervisor. Note also that CC can only provide a definitive *unsat* answer (no view of non-array constraints) while FD can provide both definitive *sat* and *unsat* answers.

The list of critical pairs is dynamically modified by the supervisor: new pairs are added when new *select* are deduced by CC, and already proved (dis-)equal pairs are removed. In our current implementation, the supervisor queries FD on all active critical pairs at once. Querying takes place at the end of the initial propagation step, and after each labelling choice.

We consider labelling in the form of $X = k$ or $X \neq k$. The labelling procedure constrains only FD: it appears that flooding CC with all the new (dis)-equalities at each choice point was expensive and mostly worthless. In a sense, most labelling choices do not impact CC, and those which really matter are *in fine* transmitted through queries about critical pairs.

While the approach requires a dedicated implementation of the supervisor and CC (yet, most of CC is standard and easy to implement), any CP(FD) solver can be used in black-box, as long as it provides support for the atomic constraints considered and the two functions `is_fd_eq(x,y)` and `is_fd_diff(x,y)`, stating if two variables can be proved equal or different within the current domain information. These two functions are either available or easy to implement in most CP(FD) systems. Support for ALLDIFFERENT is a plus, but not mandatory.

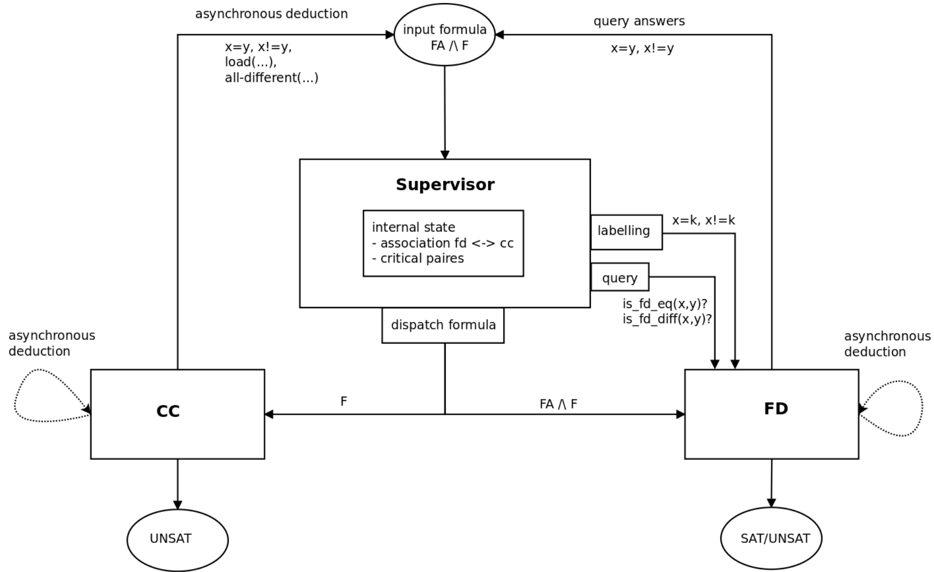


Fig. 3. Detailed view of the communication mechanism

Theoretical properties. Properties of FDCC are summarised in the next theorem. A filtering algorithm is correct if it does not discard any domain value participating into a solution of the underlying formula to solve. A decision procedure is said to be correct if both positive and negative results can be trusted, and complete if it terminates.

Theorem 1. *Assuming that* FD *filtering algorithm is correct, then* FDCC *is correct and complete.*

4.5 Running examples

Consider the array formulas extracted from Fig. 1. FD solves each formula in less than 1 second. For Prog1, CC immediately determines that (1) is *unsat*, as $i = j$ allows to merge e and f, which are declared to be different. For Prog2, in CC, the formula is not detected as being *unsat* (the size constraint over T being not taken into account), but rule *(FC-2)* produces the new disequalities $i \neq j$, $i \neq k$ and $j \neq k$. Then, the two cliques (e, f, g) and (i, j, k) are identified. In FD, the domains of i, j, k are pruned to 0..1 and *local* filtering alone cannot go further. However, when considering the cliques previously identified, two supplementary *global constraints* are added to the constraint store: ALLDIFFERENT(e, f, g) and ALLDIFFERENT(i, j, k). The latter and the pruned domains of i, j, k allow FDCC to determine that (2) is *unsat*.

5 Implementation and experimental results

5.1 Implementation of FDCC

We developed a prototype constraint solver implementing the FDCC approach. FDCC is a constraint solver over the theory of arrays augmented with finite domains arithmetic. It takes as input formulas written in the given theory and classifies them as being *sat* or *unsat*. In the former case, the tool also returns a solution under the form of a complete instantiation of the variables. Formulas may include array select and store, array size declaration, variable equalities and disequalities, finite domains specifications and arithmetic constraints on finite domain variables.

FDCC is implemented in SICStus Prolog and is about 1.7 KLOC. It exploits the SICStus `clpfd` library [9] which provides an optimised implementation of ALLDIFFERENT as well as efficient filtering algorithms for arithmetical constraints over FD. The FD solver is extended with our own implementations of the array select and store operations [10]. We use simple labelling heuristics such as *first-fail* and *first-fail constraint* [9]. Communication is implemented through message passing and awakenings. ALLDIFFERENT constraints are added each time a 3-clique is detected. Restricting clique computations to 3-cliques is advantageous to master the combinatorial explosion of a more general clique detection. Of course, more interesting deductions may be missed (e.g. 4-cliques) but we hypothesise that these cases are rare in practise. The 3-clique detection is launched each time a new disequality constraint is considered in CC.

CPU runtime is measured on an Intel Pentium 2.16GHZ machine running Windows XP with 2.0GB of RAM.

5.2 Experimental evaluation on random instances

Using randomly generated formulas is advantageous for evaluating an approach, as there is no bias in the choice of problems. However, there is also a threat to validity as random formulas might not fairly represent reality. In SAT-solving, it is well known that solvers that perform well on randomly generated formulas are not necessary good on real-world problems. To mitigate the risk, we built a dedicated random generator that produces easy-to-solve as well as hard-to-solve instances.

Formula generation. We distinguish *four different classes of formulas*, depending on whether linear arithmetic constraints are present or not (in addition to array constraints) and whether array constraints are (a priori) “easy” or “hard”. Easy array constraints are built upon three arrays, two without any *store* constraint, and the third created by two successive stores. Hard array constraints are built upon 6 different arrays involving long chains of store (up to 8 successive stores to define an array). The four classes are:

- AEUF-I (easy array constraints),
- AEUF-II (hard array constraints),

– AEUF+LIA-I (easy array constraints plus linear arithmetic),
– AEUF+LIA-II (hard array constraints plus linear arithmetic).

We performed two distinct experiments: in the first one we try to *balance sat and unsat formulas* and *more or less complex-to-solve formulas* by varying the formulas length, around and above the *complexity threshold*, while in the second experiment, we regularly increase the formula length in order to cross the *complexity threshold*. Typically, in both experiments, small-size random formulas are often easy to prove *sat* and large-size random formulas are often easy to prove *unsat*. In our examples, formula length varies from 10 to 60.

The other parameters are the following: formulas contain around 40 variables (besides arrays), arrays have a size of 20 and all variables and arrays range over domain 0..50. Interestingly, we also ran experiments with domains in 0..1000 and results were not significantly different.

Properties to evaluate. We are interested in two different aspects when comparing two solvers: (1) the ability to solve as many formulas as possible, and (2) the average computation time on easy formulas.

These two properties are both very important in a verification setting: we want of course to solve a high ratio of formulas, but a solver able to solve many formulas with an important overhead may be less interesting in some contexts than a faster solver missing only a few difficult-to-solve formulas.

Competitors. We submitted the formulas to three versions of FDCC. The first version is the standard FDCC described so far. The second version includes only the CC algorithm while the third version implements only the FD approach. In addition, we use also two witnesses, HYBRID and BEST. HYBRID represents a naive concurrent (black-box) combination of CC and FD: both solvers run in parallel, the first one getting an answer stops the other. BEST simulates a portfolio procedure with "perfect" selection heuristics: for each formula, we simply take the best result among CC and FD. BEST and HYBRID are not implemented, but deduced from results of CC and FD.

All versions are correct and complete, allowing a fair comparison. The CC version requires that the labelling procedure communicates each (dis-)equality choice to CC in order to ensure correctness.

We are primarily interested in comparing FDCC to FD since we want to improve over current CP(FD) handling of arrays. CC and HYBRID serve as witnesses, in order to understand if our combination goes further in practise than just a naive black-box combination. Finally, BEST serves as a reference point, representing the best possible black-box combination.

Results of the first experiment. For each formula, a time-out of 60s was positioned. We report the number of *sat*, *unsat* and *timeout* answers for each solver in Tab. 1.

As expected for pure array formulas (AEUF-*), FD is better on the *sat* instances, and CC behaves in an opposite way. Performance of CC decrease quickly on hard-to-solve *sat* formulas. Surprisingly, the two procedures behave quite differently in presence of arithmetic constraints: we observe that *unsat* formulas

	AEUF-I (79)				AEUF-II (90)				AEUF+LIA-I (100)				AEUF+LIA-II (100)				total (369)			
	S	U	TO	T	S	U	TO	T	S	U	TO	T	S	U	TO	T	S	U	TO	T
CC	26	37	16	987	2	30	58	3485	1	21	78	4689	0	27	73	4384	29	115	225	13545
FD	39	26	14	875	35	18	37	2299	50	47	3	199	30	60	10	622	154	151	64	3995
FDCC	**40**	37	**2**	**144**	**51**	30	**9**	**635**	**52**	48	**0**	**24**	**38**	60	**2**	**154**	**181**	175	**13**	**957**
BEST	39	37	3	202	35	30	25	1529	50	48	2	139	30	60	10	622	154	175	40	2492
HYBRID	39	37	3	242	35	30	25	1561	50	48	2	159	30	60	10	647	154	175	40	2609

S : # sat answer, U : # unsat answer, TO : # time-out (60 sec), T: time in sec.

Table 1. Experimental results of the first experiment

become often easily provable *with domain arguments*, explaining why FD performs better and CC worst compared to the AEUF-* case.

Note that computation times reported in Tab. 1 are dominated by the number of time-outs, since here solvers often succeed quickly or fail. Hence BEST and HYBRID do not show any significant difference in computation time, while in case of success, BEST is systematically 2x faster than HYBRID.

The experiments show that:

- FDCC *solves strictly more formulas* than FD or CC taken in isolation, and even more formula than BEST. Especially, there are 22 formulas solved only by FDCC, and FDCC shows 5x less TO than FD and 3x less TO than BEST.
- FDCC *has only a very affordable overhead* over CC and FD when they succeeds. Actually, FDCC was at worst 4x slower than CC or FD, and on average around 1.5x slower. While we do not provide a full comparison for each formula, one can estimate the overhead of FDCC as follow: given a formula category, take computation times t and t' for FDCC and BEST, and d the difference between #TO in FDCC and #TO in BEST. Then for each category, $t \approx t' - 60 \times d$ (for the whole set of formulas, we found a 1.1x ratio).
- These two results have been observed for the four classes of programs, for both *sat* and *unsat* instances, and a priori easy or hard instances. Hence, FDCC *is much more robust* than FD or CC are.

Results of the second experiment. In this experiment, 100 formulas of class AEUF-II are generated with length l, l varying from 10 to 60. While crossing the *complexity threshold*, we record the number of time-outs (60sec). In addition, we used two metrics to evaluate the capabilities of FDCC to solve formulas, **Gain** and **Miracle**:

- **Gain** is defined as follows: each time FDCC classifies a formula that none of (resp. only one of) CC and FD can classify, **Gain** is rewarded by 2 (resp. 1); each time FDCC cannot classify a formula that one of (resp. both) CC and FD can classify, **Gain** is penalised by 1 (resp. 2). Note that the -2 case never happens during our experiments.

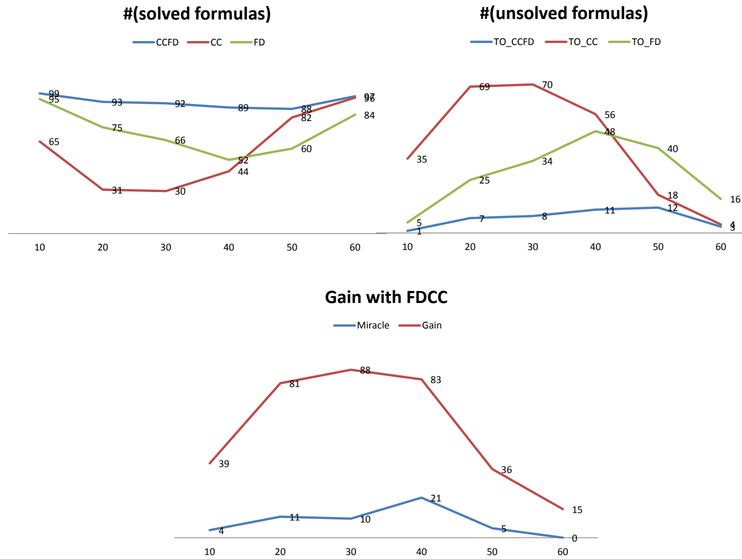


Fig. 4. Experimental results for the 2nd experiment

– **Miracle** is defined as the number of times FDCC gives a result while both CC and FD fail to do so.

Fig. 4 shows the number of solved formulas for each solver, the number of formulas which remain unsolved because of time out, and both the values of *Gain* and *Miracles*. We see that the number of solved formulas is always greater for FDCC (about 20% more than FD and about 70% more than CC). Moreover, FDCC presents maximal benefits for formula lengths in between 20 and 40, i.e. for lengths close to the complexity threshold, meaning that relative performance are better on hard-to-solve formulas. For these lengths, the number of unsolved formulas is always less than 11 with FDCC, while it is always greater than 25 with both CC and FD.

Conclusion. Experimental results show that FDCC performs better than FD and CC taken in isolation, especially on hard-to-solve formulas, and is very competitive with portfolio approaches mixing FD and CC. Especially, FDCC solves strictly more formulas than its competitors (3x less TO than BEST) and shows a reasonable overhead (1.1x average ratio vs BEST). Moreover, relative performance are better on hard-to-solve formulas than on easy-to-solve formulas, suggesting that it becomes especially worthwhile to combine global symbolic reasoning with local filtering when hard instances have to be solved. Finally, FDCC performance seems to be robust to the class of formulas considered.

This is particularly interesting in a verification setting, since it means that FDCC can be clearly preferred to the standard FD-handling of arrays in any

context, i.e. whether we want to solve a few complex formulas or we want to solve as many as formula in a small amount of time.

6 Related work

It is well known in the SMT community that solving formulas over arrays and integer arithmetic in an efficient way through a Nelson-Oppen combination framework [22] is difficult. Indeed, since arrays and (linear) integer arithmetic are *non convex theories*, NO requires to communicate all *implied disjunctions of equalities* to ensure correctness. Such a propagation may be much more expensive than satisfiability check [2]. NO with delayed theory combination [1, 2] requires only the propagation of implied equalities, at the price of adding new boolean variables for all potential equalities between variables. Some works aim at mitigating the potential overhead of these extra-variables, for example the model-based combination implemented in Z3 [12], where equalities are propagated lazily. Another possibility is to reduce the theory of arrays to the theory of equality by systematic "inlining" of axioms (4) and (5) to remove all *store* operators, at the price of introducing many case-splits. The encoding can be eager [18] or lazy [8].

Filtering approaches for array constraints are already discussed in Section 3. The ELEMENT constraints and disjunctions can express update constraints. However, a dedicated update constraint is more efficient in case of non-constant indexes. The work of Beldiceanu et al. [5] has shown that it is possible to capture global state of several ELEMENT constraints with an automaton. Our approach is more general as it handles any possible combination of ELEMENT (and UPDATE) constraints but it is also only symbolic and thus less effective. In our framework, the CC algorithm cannot prune the domain of index or indexed variables. In fact, our work has more similarities with what has been proposed by Nieuwenhuis on his DPLL(ALLDIFFERENT) proposition[5]. The idea is to benefit from the efficiency of several global constraints in the DPLL algorithm for SAT encoded problems. In FDCC, we derive ALLDIFFERENT global constraints from the congruence closure algorithm for similar reasons. Nevertheless, our combined approach is fully automated, which is a keypoint to address array constraint systems coming from various software verification problems.

Several possibilities can be considered to implement constraint propagation when multiple propagators are available [26]. First, an external solver can be embedded as a new global constraint in FD, as done for example on the QUAD global constraint [19]. This approach offers global reasoning over the constraint store. However, it requires fine control over the awakening mechanism of the new global constraint. A second approach consists in calling both solvers in a concurrent way. Each of them is launched on distinct threads, and both threads prune a common constraint store that serves of blackboard. This approach has been successfully implemented in Oz [27]. The difficulty is to identify which information must be shared, and to do it efficiently. A third approach consists

[5] `http://www.lsi.upc.edu/ roberto/papers/CP2010slides.pdf`

in building a master-slave combination process where one of the solvers (here CC) drives the computation and call the other (FD). The difficulty here is to understand when the master must call the slave.

We follow mainly the second approach, however a third agent (the supervisor) acts as a lightweight master over CC and FD to synchronise both solvers through queries.

7 Conclusions and perspectives

This paper describes an approach for solving conjunctive quantifier-free formulas combining arrays and finite-domain constraints over indexes and elements (typically, bounded arithmetic). We sketch an original decision procedure that combines ideas from symbolic reasoning and finite-domain constraint solving. The bi-directional communication mechanism proposed in this paper lies on the opportunity of improving the deductive capabilities of the congruence closure algorithm with finite domains information. We also propose ways of keeping the communication overhead tractable. Experiments show that our combined approach performs better than any portfolio-like combination of a symbolic solver and a filtering-based solver. Especially, our procedure enhances greatly the deductive power of standard CP(FD) approaches for arrays. In future work, we plan to incorporate our algorithm into a CP(FD)-based verification tool in order to evaluate its benefits on real-life problems.

Acknowledgements. We are very grateful to Pei-Yu Li who proposed a preliminary encoding of FDCC during her trainee period, and Nadjib Lazaar for comparative experiments with OPL.

References

1. Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Silvio Ranise, Peter van Rossum, Roberto Sebastiani. Efficient Satisfiability Modulo Theories via Delayed Theory Combination. In CAV 2005. Springer (2005)
2. Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Roberto Sebastiani. Delayed theory combination vs. Nelson-Oppen for satisfiability modulo theories: a comparative analysis. In Ann. Math. Artif. Intell., vol 55 (1-2), 2009
3. Sébastien Bardin and Philippe Herrmann. Structural testing of executables. In *1th Int. Conf. on Soft. Testing, Verif. and Valid. (ICST'08)*, pages 22–31, 2008.
4. Sébastien Bardin, Philippe Herrmann and Florian Perroud. An alternative to sat-based approaches for bit-vectors. In *Tools and Algorithms for the Construction and Analysis (TACAS'10)*, pages 84–98, 2010.
5. Nicolas Beldiceanu, Mats Carlsson, Romuald Debruyne, and Thierry Petit. Reformulation of global constraints based on constraints checkers. *Constraints*, 10:339–362, October 2005.
6. Bernard Botella, Arnaud Gotlieb and Claude Michel. Symbolic execution of floating-point computations. *The Software Testing, Verification and Reliability journal*, 16(2):pp 97–121, June 2006.

7. Sebastian Brand. Constraint propagation in presence of arrays. *Computing Research Repository*, 2001. 6th Workshop of the ERCIM Working Group on Constraints, 2001.
8. Robert Brummayer and Armin Biere. Lemmas on demand for the extensional theory of arrays. In *SMT '08/BPR '08*, pages 6–11. ACM, 2008.
9. Mats Carlsson, Greger Ottosson, and Björn Carlson. An open–ended finite domain constraint solver. In *Proc. of Programming Languages: Implementations, Logics, and Programs*, 1997.
10. Florence Charreteur, Bernard Botella and Arnaud Gotlieb. Modelling dynamic memory management in constraint-based testing. *The Journal of Systems and Software*, 82(11):1755–1766, Nov. 2009. Special Issue: TAIC-PART 2007 and MUTATION 2007.
11. Hélène Collavizza, Michel Rueher and Pascal Van Hentenryck. Cpbpv: A constraint-programming framework for bounded program verification. In *Proc. of CP2008*, LNCS 5202, pages 327–341, 2008.
12. Leonardo de Moura and Nikolaj Bjørner. Model-based theory combination. *Electron. Notes Theor. Comput. Sci.*, 198(2):37–49, 2008.
13. Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *TACAS'08/ETAPS'08: Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, pages 337–340. Springer-Verlag, 2008.
14. Peter J. Downey and Ravi Sethi. Assignment commands with array references. *J. ACM*, 25:652–666, October 1978.
15. Arnaud Gotlieb, Bernard Botella, and Michel Rueher. A clp framework for computing structural test data. In *Proceedings of Computational Logic (CL'2000)*, LNAI 1891, pages 399–413, London, UK, July 2000.
16. Arnaud Gotlieb, Michel Leconte, and Bruno Marre. Constraint solving on modular integers. In *Proc. of the 9th Int. Workshop on Constraint Modelling and Reformulation (ModRef'10), co-located with CP'2010*, St Andrews, Scotland, Sept. 2010.
17. Pascal Van Hentenryck and Jean-Philippe Carillon. Generality versus specificity: An experience with ai and or techniques. In *Proc. of AAAI'88, AAAI Press/The MIT Press*, pages 660–664, 1988.
18. Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. 2008.
19. Yahia Lebbah, Claude Michel, Michel Rueher, and David Daney. Efficient and safe global constraints for handling numerical constraint systems. *SIAM J. Numer. Anal*, 42:2076–2097, 2005.
20. Bruno Marre and Benjamin Blanc. Test selection strategies for lustre descriptions in gatel. *Electronic Notes in Theoretical Computer Science*, 111:93 – 111, 2005.
21. Laurent Michel and Pascal Van Hentenryck. *Constraint-Based Local Search*. MIT Press, 2005.
22. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1:245–257, October 1979.
23. Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.
24. Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *Proc. of the twelfth national conference on Artificial intelligence (vol. 1)*, AAAI '94, pages 362–367, 1994.
25. John Rushby. Verified software: Theories, tools, experiments. chapter Automated Test Generation and Verified Software, pages 161–172. Springer-Verlag, 2008.

26. Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, December 2008.
27. Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz, and Christian Schulte. Logic programming in the context of multiparadigm programming: the Oz experience. *Theory and Practice of Logic Programming*, 3(6):715–763, November 2003.