A Write-Based Solver for SAT Modulo the Theory of Arrays

Miquel Bofill
Universitat de Girona

Robert Nieuwenhuis
Albert Oliveras
Enric Rodríguez-Carbonell
and Albert Rubio
Universitat Politècnica de Catalunya

***Abstract*—The extensional theory of arrays is one of the most important ones for applications of SAT Modulo Theories (SMT) to hardware and software verification.**

Here we present a new T-solver for arrays in the context of the DPLL(T) approach to SMT. The main characteristics of our solver are: (i) no translation of writes into reads is needed, (ii) there is no axiom instantiation, and (iii) the T-solver interacts with the Boolean engine by asking to split on equality literals between indices.

As far as we know, this is the first accurate description of an array solver integrated in a state-of-the-art SMT solver and, unlike most state-of-the-art solvers, it is not based on a lazy instantiation of the array axioms. Moreover, it is very competitive in practice, specially on problems that require heavy reasoning on array literals.

I. Introduction

Over the last few years, traditional generic proof-search methods have been increasingly replaced by more efficient domain-specific procedures or procedures for fragments of certain logics. Despite being more specific, these procedures have seen its way into many real-world applications since most problems can be decomposed, either manually or automatically, into smaller problems to which these concrete procedures can be applied.

Among them, *Satisfiability Modulo Theories* (SMT) tools have received special attention. These procedures decide the satisfiability of (usually ground) formulas modulo some background theory T. The choice of the theory depends on the particular application area: when verifying complex designs, the theory of *Equality with Uninterpreted Functions (EUF)* comes in handy since it allows one to abstract away uninteresting or too complicated details of the system to be verified [1]; if, on the other hand, one is interested in verifying low-level aspects of, e.g., a microprocessor, the theory of *bit-vectors* provides the necessary level of detail [2], [3]; and, just to give another example, for the verification of timed automata a certain fragment of linear arithmetic called *Difference Logic* is the most appropriate choice [4]. Hence, SMT tools deal with problems that may consist of thousands of clauses like:

$p \ \vee \ a{=}f(b{-}c) \ \vee \ read(s,\, f(b{-}c)\,){=}d \ \vee \ a{-}g(c) \,{\leq}7$

containing purely propositional atoms as well as atoms over (combined) theories.

One of the main approaches to SMT has been called DPLL(T) [5], which consists of a general DPLL(X) engine, very similar in nature to a SAT solver, that works in cooperation with a theory solver $Solver_T$. In the most basic version of this approach the engine is in charge of enumerating propositional models of the formula, whereas $Solver_T$ is responsible for checking whether these models are consistent with the theory T (e.g. if T is the theory of linear arithmetic and the current Boolean model contains $x{+}2y \leq 0$, $-y - \frac{1}{2}z \leq 0$ and $x - z > 1$, then $Solver_T$ has to detect that the current assignment is T-inconsistent).

In this paper, we present a new $Solver_T$ for the *Theory of Arrays with Extensionality*. This theory is useful for both software and hardware verification, since it can be used to model the behavior of, e.g., arrays and memories. The signature of the theory consists of two interpreted function symbols: *read*, used to retrieve the element stored at a certain index of the array, and *write*, used to modify an array by updating the element stored at a certain index. More formally, their behavior can be modeled with the following axioms:

$$\begin{array}{l} \forall a : Array, \forall i, j : Index, \forall x : Value \\ \quad i{=}j \quad \Rightarrow \quad read(write(a,i,x),j) = x \\ \quad i{\neq}j \quad \Rightarrow \quad read(write(a,i,x),j) = read(a,j) \end{array}$$

Finally, one still has to consider the extensionality property, stating that two arrays that coincide on all indices are indeed equal. This is enforced by the axiom:

$$\begin{array}{l} \forall a, b : Array \\ \quad (\forall i : Index \ \ read(a,i) = read(b,i)) \quad \Rightarrow \quad a = b \end{array}$$

A possible approach to decide the satisfiability of ground literals in the theory of arrays with extensionality is to consider the necessary instances of the aforementioned axioms and let them be handled by the DPLL(X) engine in cooperation with a $Solver_T$ for EUF. In this setting, the array solver is reduced to a module that only generates clauses. This approach is used in state-of-the art SMT solvers like Yices [6] and Z3 [7] but, as far as we know, there is no precise description of it in the literature.

Our approach does not follow the same lines but is rather based on a careful analysis of the problem that allows us to infer, for each array, which are the relevant indices and which values are stored at these indices. Once this information has been made transparent, checking satisfiability of sets of literals

becomes an easy task. Unlike the approach of [8], we do not remove significant *write*'s from the problem, and hence there is no need to use the notion of "equality of arrays except in a certain set of indices" used in [8].

As usual, the extensionality axiom is addressed by forcing that, if two arrays are different, there is an index, called a *disequality witness*, where the two arrays do not coincide. For efficiency reasons, the introduction of these witnesses is delayed so as to reduce the combinatorial explosion caused by the comparison between indices. This explosion is also mitigated in our solver by identifying situations where the set of literals can be proved to admit a model without the need to construct a total partition over the sets of values and indices.

These ideas lead to a very natural and easy-to-understand solver. As a result, all proofs also become noticeably intuitive. Moreover, using the adequate strategy, our solver performs very well on problems that require heavy reasoning on array literals.

The rest of the paper is structured as follows. In Section II we give basic definitions and notation. Then, in Section III we describe our solver in terms of transformation rules and prove their correctness. After that, in Section IV we give some details of the integration of the solver in a DPLL(T) system. Finally, we present experimental results in Section V and we conclude in Section VI.

II. Preliminaries

We will consider a set of index constants denoted by $\mathcal{I}$ and a set of value constants denoted by $\mathcal{V}$. An array expression is either an array constant or $write(A, i, x)$ where A is an array expression, i is an index constant and x is a value constant. Arrays of the form $write(\ldots write(a, i_1, x_1) \ldots, i_n, x_n)$ will be written as $write(a, \langle i_1, \ldots, i_n\rangle, \langle x_1, \ldots, x_n\rangle)$, and as $write(a, \vec{I}, \vec{X})$ for short. Note that the first elements of $\vec{I}$ and $\vec{X}$ correspond to the innermost *write* and the last elements to the outermost *write*. We are using vector notation for $\vec{I}$ and $\vec{X}$ to emphasize that the elements are ordered and to refer to the elements at position p as i_p and x_p, respectively. Given $\vec{I}$, we will write I to denote the set of elements in $\vec{I}$. As a convention, the array constant a will be seen, when necessary, as $write(a, \emptyset, \emptyset)$.

In what follows, possibly subscripted a, b, c denote array constants and A, B, C denote array expressions. When we write $\equiv$ we mean syntactic equality between expressions and $=_{set}$ denotes equality between sets.

An *array satisfaction (ASAT) problem* is a conjunction of literals of the form:

- $\bot$ (representing an unsatisfiable formula).
- $x = read(A, i)$ or $x \neq read(A, i)$, where A is an array expression, $i \in \mathcal{I}$ and $x \in \mathcal{V}$.
- $A = B$ or $A \neq B$ where A and B are array expressions.
- $x = y$ or $x \neq y$, where $x, y \in \mathcal{I}$ or $x, y \in \mathcal{V}$.

A. The Extensional Theory of Arrays

The extensional theory of arrays is defined by the following axioms:

read-write axioms:

$$\begin{array}{l}\forall a : Array, \forall i, j : Index, \forall x : Value \\ \quad i{=}j \;\Rightarrow\; read(write(a, i, x), j) = x \\ \quad i{\neq}j \;\Rightarrow\; read(write(a, i, x), j) = read(a, j)\end{array}$$

extensionality axiom:

$$\begin{array}{l}\forall a, b : Array \\ \quad (\forall i : Index \;\; read(a, i) = read(b, i)) \;\Rightarrow\; a = b\end{array}$$

The following well-known axioms are not necessary if we consider extensionality, since they are entailed by the previous axioms:

write-write axioms:

$$\begin{array}{l}\forall a : Array, \forall i, j : Index, \forall x, y : Value \\ \quad i{=}j \Rightarrow write(write(a, i, x), j, y){=}write(a, j, y) \\ \quad i{\neq}j \Rightarrow write(write(a, i, x), j, y){=}write(write(a, j, y), i, x)\end{array}$$

Note that, however, the axioms of read-write and write-write do not entail the extensionality axiom.

Example 1: Let us consider the following set of literals:

$$\begin{array}{lll} write(a, i, x){\neq}b & read(b, i){=}y & read(write(b, i, x), j){=}y \\ \quad a{=}b & & \quad\quad i{=}j \end{array}$$

The two literals on the left imply that $read(b, i) \neq x$. This can be seen by replacing a by b in the first disequality, applying a $read$ at position i at both sides of the disequality and using the first read-write axiom. The two literals on the right imply that $x = y$. This can be deduced if we use that $i = j$ and apply the first read-write axiom at the top-right hand equation. Together with the equation in the middle we get that $read(b, i) = x$ which shows that the set of literals is T-inconsistent. □

B. Models

The models we will consider are given by a mapping I_I from $\mathcal{I}$ to some domain $D_{\mathcal{I}}$, a mapping I_V from $\mathcal{V}$ to some domain $D_{\mathcal{V}}$ and a mapping I_A from array constants to functions $f_A : D_{\mathcal{I}} \longrightarrow D_{\mathcal{V}}$, s.t. if $I_A(a) = f_a$ then $I_A(write(a, i, x)) = f'_a$ where $f'_a(I_I(i)) = I_V(x)$ and $f'_a(k) = f_a(k)$ for all $k \in D_{\mathcal{I}}$ such that $k \neq I_I(i)$. Finally, the interpretation on values is extended to the $read$ operation by the equality $I_V(read(a, i)) = I_A(a)(I_I(i))$.

Let M be a set of equalities and disequalities over $\mathcal{I}$ or $\mathcal{V}$. A position p in $\vec{I}$ is an *M-significant position in $\vec{I}$* if $M \models i_p \neq i_k$ for all $k > p$. The *set of M-significant indices in $\vec{I}$* is denoted by $sig_M(\vec{I})$ and is defined as the set of those indices i_p in I such that p is an M-significant position in $\vec{I}$.

Example 2: Let $M = \{j \neq l, l = k\}$ and $\vec{I} = (i, j, k, l)$. The set $sig_M(\vec{I})$ is $\{j, l\}$. Note that k is not an M-significant index because $M \models k{=}l$ and i is not an M-significant index because, among other reasons, $M \not\models i{\neq}j$. If we now take M and $\vec{I'} = (i, j, k, l, i)$ then i is now an M-significant index in $\vec{I'}$ because of its last occurrence. □

The pairs $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are said to be M-equivalent, if (i) for all $i, i' \in I$ (similarly for J), we have $M \models i = i'$ or $M \models i \neq i'$, (ii) $M \models sig_M(\vec{I}) =_{set} sig_M(\vec{J})$, and (iii)

for all $i_p \in sig_M(\vec{I})$ and $j_q \in sig_M(\vec{J})$, if $M \models i_p = j_q$ then $M \models x_p = y_q$.

III. A New Solver for the Extensional Theory of Arrays

In this section we present, in terms of transformation rules, a solver for deciding the satisfiability of ASAT problems. Our transformation system deals with conjunctions of literals, but they will be represented as $M \mid P$, so as to explicitly distinguish the subset M of literals than can be treated by a simple Union-Find algorithm, i.e., (dis)equalities between indices or values, from the conjunction P of those literals involving arrays. As expected, $M \mid P$ has to be understood as $M \wedge P$. Given a conjunction of literals P, by $P\{a := A\}$ we mean the result of replacing all occurrences of a by A in P.

Definition 1: The system of transformation rules in Figure 1 is called $\mathcal{A}$.

Let us briefly comment on some of the rules. First of all, for rules like **True equality**, **Array inconsistency** or **Disequality witness introduction**, recall that an array constant a can be expressed as $write(a, \emptyset, \emptyset)$.

The **Significance query** rule is used to detect the set of significant indices in an array expression, whereas the **Equality index query** rule is used to detect equal indices in both sides of an equality.

In the **Write introduction** rule, by propagating $b := write(c, i, x)$ in the conclusion we make explicit that b is an array with x at position i. Note that we could keep that literal in order to be able to later recover a model for the original set of array constants. This is also the case for the **Substitution** rule.

Finally, let us mention that the **Disequality witness introduction** rule takes care of extensionality: if we have $write(a, \vec{I}, \vec{X}) \neq write(b, \vec{J}, \vec{Y})$ and we know that the difference is not in the significant indices in $\vec{I}$ (which coincides with significant indices in $\vec{J}$), then we can infer that a and b are different at some position not in $\vec{I}$. The introduction of these witnesses can be delayed at convenience. Similarly to **Write introduction** and **Substitution** we could keep the two substitutions as literals so as to later recover a model for the original problem.

As we will show below in all these rules the premise and the disjunction of the conclusions are equisatisfiable. Hence, they can be used to produce a *derivation tree* rooted by the original array problem and where every non-terminal node has as children the conclusions of an application of a transformation rule. As we will see, the problem will be unsatisfiable if and only if all terminal nodes are $\bot$.

Lemma 1: For all rules in $\mathcal{A}$ the premise and the disjunction of its conclusions are equisatisfiable.

Proof: We show that there is a model for the premise if and only if there is a model for the disjunction of the conclusions. We only detail the proof for the following two rules:

- **Write introduction**. For the left to right implication, let (I_I, I_V, I_A) be a model for $M \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})$. Since by the conditions of the transformation rule, $I_I(i_p) \neq I_I(i_k)$ for all $k > p$, and $I_I(i_p) \neq I_I(j)$ for every $j \in \vec{J}$, we have that $I_A(b)(I_I(i_p)) = I_V(x_p)$. Therefore, extending I_A to interpret the fresh constant c as $I_A(c) = f_c$, where $f_c(k) = (I_A(b))(k)$ for all $k \in D_{\mathcal{I}}$, we have a model for $M \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y}) \wedge b = write(c, i_p, x_p)$ and hence also of the conclusion. For the right to left implication, since b does not appear in the conclusion we only have to extend the model so that b is interpreted as $write(c, i_p, x_p)$.
- **Array inconsistency**. In this case we show that there cannot be any model for $M \mid write(a, \vec{I}, \vec{X}) \neq write(a, \vec{J}, \vec{Y})$ if $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are M-equivalent. Assume we have a model for M. Then $I_I(sig_M(\vec{I}))$ is equal to $I_I(sig_M(\vec{J}))$ and for all M-significant $i_p \in \vec{I}$ and $j_q \in \vec{J}$, if $I_I(i_p) = I_I(j_q)$ then $I_V(x_p) = I_V(y_q)$, which implies that any array interpretation must interpret $write(a, \vec{I}, \vec{X})$ and $write(a, \vec{J}, \vec{Y})$ to the same function, and thus cannot satisfy $write(a, \vec{I}, \vec{X}) \neq write(a, \vec{J}, \vec{Y})$. □

■

Definition 2: An ASAT problem is in *solved form* if either it is $\bot$ or it is $M \mid P$ where M is consistent and all literals in P are of the form:

1) $write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})$
   with $a \not\equiv b$, and $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ being M-equivalent.
2) $write(a, \vec{I}, \vec{X}) \neq write(b, \vec{J}, \vec{Y})$
   with some M-significant position p in $\vec{I}$ such that $M \not\models i_p = j$ for every $j \in \vec{J}$.
3) $write(a, \vec{I}, \vec{X}) \neq write(b, \vec{J}, \vec{Y})$
   with some M-significant position p in $\vec{I}$ and some M-significant position q in $\vec{J}$ such that $M \models i_p = j_q$ and $M \not\models x_p = y_q$.
4) $x \neq read(a, i)$
   being a an array constant.

Lemma 2: An ASAT problem in solved form is satisfiable if and only if it is not $\bot$.

Proof: The left to right implication is trivial. For the other one, let us consider a problem $M \mid P$ in solved form. First of all, we introduce a new value constant d. Then, we consider $M' = M \cup \{x \neq y \mid M \not\models x = y\}$ that, since M is consistent and equality is a convex theory[1], is also consistent. Essentially, we have extended M by adding that all what is not known to be equal is set to distinct (including all comparisons with the new constant d).

Now, let (I_I, I_V) be a model of M'. We call $f_d : D_{\mathcal{I}} \to D_{\mathcal{V}}$ the constant function that always returns $I_V(d)$. Note that, by assumption, $I_V(d)$ is different from the interpretation of all other value constants. Now, we define our array interpretation I_A as $I_A(a) = f_d$ for every array constant a. Note that, if in the rules **Substitution**, **Write Introduction** and **Disequality Witness Introduction** we had kept the substitution as a literal

[1]This means that if E is a set of equalities and disequalities, $E \models x_1 = y_1 \vee \ldots \vee x_n = y_n$ iff $E \models x_i = y_i$ for some i in $1 \ldots n$.

UF-Dispatch:

$$\frac{M \mid x \bowtie y \wedge P}{M \cup \{x \bowtie y\} \mid P}$$

with $\bowtie \in \{=, \neq\}$

Read2Write:

$$\frac{M \mid P[x = read(a, i)]}{M \mid P[a = write(b, i, x)]}$$

if a and b are array constants and b is *fresh*.

Read-Write:

$$\frac{M \mid P[read(write(A, i, x), j)]}{M \cup \{i = j\} \mid P[x] \qquad M \cup \{i \neq j\} \mid P[read(A, j)]}$$

Substitution:

$$\frac{M \mid P \wedge a = A}{M \mid P\{a := A\}}$$

if a does not occur in A.

True equality:

$$\frac{M \mid P \wedge write(a, \vec{I}, \vec{X}) = write(a, \vec{J}, \vec{Y})}{M \mid P}$$

if $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are M-equivalent.

Array inconsistency:

$$\frac{M \mid P \wedge write(a, \vec{I}, \vec{X}) \neq write(a, \vec{J}, \vec{Y})}{\bot}$$

if $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are M-equivalent.

UF-inconsistency:

$$\frac{M \mid P}{\bot} \qquad \text{if } M \text{ is inconsistent}$$

Write introduction:

$$\frac{M \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})}{M \mid (P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y}))\{b := write(c, i_p, x_p)\}}$$

if c is *fresh* and p is an M-significant position in $\vec{I}$ and $M \models i_p \neq j$ for every $j \in \vec{J}$.

Significance query:

$$\frac{M \mid P[write(write(A, \vec{I}, \vec{X}), j, y)]}{M \cup \{i = j\} \mid P[write(write(A, \vec{I}, \vec{X}), j, y)] \quad M \cup \{i \neq j\} \mid P[write(write(A, \vec{I}, \vec{X}), j, y)]}$$

if $M \not\models i = j$ and $M \not\models i \neq j$ for some $i \in \vec{I}$.

Equality index query:

$$\frac{M \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})}{M \cup i_p = j_q \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y}) \quad M \cup i_p \neq j_q \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})}$$

if p is an M-significant position in $\vec{I}$, q is an M-significant position in $\vec{J}$ and $M \not\models i_p = j_q$ and $M \not\models i_p \neq j_q$.

Equality values propagation:

$$\frac{M \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})}{M \cup x_p = y_q \mid P \wedge write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})}$$

if p is an M-significant position in $\vec{I}$, q is an M-significant position in $\vec{J}$ and $M \models i_p = j_q$ and $M \not\models x_p = x_q$.

Disequality witness introduction:

$$\frac{M \mid P \wedge write(a, \vec{I}, \vec{X}) \neq write(b, \vec{J}, \vec{Y})}{M' \mid P\{a := write(c, ni, e_1), b := write(d, ni, e_2)\}}$$

where $M' = M \cup \{ni \neq i \mid i \in \vec{I}\} \cup \{ni \neq j \mid j \in \vec{J}\} \cup \{e_1 \neq e_2\}$
if $a \not\equiv b$, $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are M-equivalent and c, d, ni, e_1 and e_2 are *fresh*.

Fig. 1. Transformation rule system $\mathcal{A}$

of the form $a = A$, where a does not occur elsewhere in P, then we would only need to extend I_A with $I_A(a) = I_A(A)$.

Let us show that (I_I, I_V, I_A) is a model of all literals in P.

1) The literal is of the form $write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})$, where $a \not\equiv b$, and $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are M-equivalent. Then, by definition of I_A we have $I_A(a) = I_A(b)$. Now, since if $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are M-equivalent then they are also M'-equivalent, we have that $I_I(sig_{M'}(\vec{I}))$ is equal to $I_I(sig_{M'}(\vec{J}))$. Moreover, for all M'-significant $i_p \in \vec{I}$ and $j_q \in \vec{J}$ such that $I_I(i_p) = I_I(j_q)$ then $I_V(x_p) = I_V(y_q)$, which implies that $I_A(write(a, \vec{I}, \vec{X})) = I_A(write(b, \vec{J}, \vec{Y}))$.
2) The literal is of the form $write(a, \vec{I}, \vec{X}) \neq write(b, \vec{J}, \vec{Y})$ where there is some M-significant position p in $\vec{I}$ such that $M \not\models i_p = j$ for every $j \in \vec{J}$. Then, since $M \not\models i_p = j$, we have that $M' \models i_p \neq j$, and hence $I_I(i_p) \neq I_I(j)$ for all j in $\vec{J}$. Now, by definition, we have that $I_A(write(a, \vec{I}, \vec{X}))$ is some function f_1 such that $f_1(I_I(i_p)) = I_V(x_p)$ and, since $I_A(b) = f_d$ and $I_I(i_p) \neq I_I(j)$ for all $j \in \vec{J}$, we have that $I_A(write(b, \vec{J}, \vec{Y}))$ is some function f_2 such that $f_2(I_I(i_p)) = I_V(d) \neq I_V(x_p)$, which implies $f_1 \neq f_2$, satisfying the literal.
3) The literal is of the form $write(a, \vec{I}, \vec{X}) \neq write(b, \vec{J}, \vec{Y})$ where there is some M-significant position p in $\vec{I}$ and some M-significant position q in $\vec{J}$ such that $M \models i_p = j_q$ and $M \not\models x_p = y_q$. Then, by definition, we have $M' \models i_p = j_q$ and $M' \models x_p \neq y_q$. Now, similarly to the previous case, we have that $I_A(write(a, \vec{I}, \vec{X}))$ is some function f_1 such that $f_1(I_I(i_p)) = I_V(x_p)$ and $I_A(write(b, \vec{J}, \vec{Y}))$ is some function f_2 such that $f_2(I_I(j_q)) = I_V(y_q)$. Therefore, since $I_I(i_p) = I_I(j_q)$ and $I_V(x_p) \neq I_V(y_q)$, we have $f_1 \neq f_2$, satisfying the literal.
4) The literal is of the form $x \neq read(a, i)$. Then, by definition, we have that $M' \models x \neq d$, and hence $I_V(x) \neq I_V(d)$. Moreover, since $I_A(a) = f_d$, then $I_V(read(a, i)) = f_d(I_I(i)) = I_V(d) \neq I_V(x)$, hence satisfying the literal. □

■

Definition 3: An *$\mathcal{A}$-derivation tree* is a tree whose nodes are ASAT problems and where the children of a node $M \mid P$ are the conclusions of a transformation rule in $\mathcal{A}$ applied to $M \mid P$. We will say that an $\mathcal{A}$-derivation tree is *solved* iff all its leafs are solved forms.

Lemma 3: All paths in a A-derivation tree are finite.

Proof: First of all we show that given an ASAT problem $M \mid P$, the set of indices and values that can occur in any $\mathcal{A}$-derivation tree rooted by $M \mid P$ is finite. This is easy to see since the only rule that introduces new constants is **Disequality witness introduction**, but since every application of this rule removes a disequality literal and no other rule introduces new disequalities, this can only happen a finite number of times. Hence, this also proves that this rule cannot be applied an infinite number of times.

Regarding the rules **Significance query**, **Equality index query** and **Equality values propagation**, they can only be applied a finite number of times. This is because they add a (dis)equality between indices or values to M that was not previously entailed by it. Since there are only a finite number of indices and values, this can only happen finitely often.

Similarly, the **Write introduction** rule can only be applied a finite number of times. The key argument is based on the two following facts: (i) the number of literals appearing in any node of the derivation tree does not increase, since no rule adds new literals and (ii) no rule removes $write$'s from an equality between array expressions. Given these two facts, if we denote by N_I the maximum number of indices that can occur in the derivation tree, we know that for every equality literal in the original problem, we can apply **Write Introduction** at most $2 * N_I$ times, since after $2 * N_I$ applications all indices will appear in both sides of the equality and the rule will no longer be applicable.

Hence, any infinite derivation should end with an infinite sequence of applications of **UF-Dispatch**, **Read-Write**, **Read2Write**, **Substitution** and **True equality**. But this cannot be the case: given an ASAT problem $M \mid P$, we can associate to it the triple of natural numbers $(\#\text{literals in } P, \#reads, |P|)$, being $|P|$ the size of P. It is an easy exercise to check that using a lexicographic ordering all remaining rules decrease that triple of natural numbers.

■

The following lemma is easily proved by comparing the conditions imposed on solved forms and the conditions imposed on the rules.

Lemma 4: If an ASAT problem is not a solved form then a transformation rule in $\mathcal{A}$ can be applied.

Proof: If M is not consistent then we can apply the **UF-inconsistency** rule. Otherwise, we proceed by case analysis according to the kind of literal that is not in solved form.

1) The literal is of the form $a = A$. If a does not occur in A then the **Substitution** rule can be applied. Otherwise, if $A \equiv a$ then **True equality** can be applied. Finally, if A is $write(a, \vec{I}, \vec{X})$ with $\vec{I} \neq \emptyset$, we can apply **Write introduction** for the last index i_n in $\vec{I}$.
2) The literal is of the form $write(a, \vec{I}, \vec{X}) = write(b, \vec{J}, \vec{Y})$, with $\vec{I}$ and $\vec{J}$ non-empty. If there is a pair of indices i_p and i_q in $\vec{I}$ (or in $\vec{J}$) such that $M \not\models i_p = i_q$ and $M \not\models i_p \neq i_q$, we can apply the **Significance query** rule. Otherwise, if there is some M-significant position p in $\vec{I}$ and some M-significant position q in $\vec{J}$ s.t. $M \not\models i_p = j_q$ and $M \not\models i_p \neq j_q$ then we can apply the **Equality index query** rule. Otherwise, if $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are not M-equivalent it is because either (i) there is some M-significant index i_p in $\vec{I}$ such that $M \models i_p \neq j$ for all j in $\vec{J}$ and we can apply **Write introduction** (the same happens if we exchange $\vec{I}$ and $\vec{J}$) or (ii) $M \models sig_M(\vec{I}) =_{set} sig_M(\vec{J})$ but there exist an M-significant position p in $\vec{I}$ and an M-significant position q in $\vec{J}$ such that $M \models i_p = j_q$ but $M \not\models x_p = y_p$ and then we can apply **Equality values propagation**.

The only possibility left now is that $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are indeed M-equivalent and then either the literal is in solved form or we can apply **True equality**.

3) The literal is of the form $write(a, \vec{I}, \vec{X}) \neq write(b, \vec{J}, \vec{Y})$. If there is a pair of indices i_p and i_q in $\vec{I}$ (or in $\vec{J}$) such that $M \not\models i_p = i_q$ and $M \not\models i_p \neq i_q$, we can apply the **Significance query** rule. Otherwise, if the literal is not in solved form, then $(\vec{I}, \vec{X})$ and $(\vec{J}, \vec{Y})$ are M-equivalent, and hence we can either apply the **Array Inconsistency** rule when $a \equiv b$, or we can apply the **Disequality witness introduction** otherwise.
4) The literal is of the form $x = read(A, i)$. If A is not a constant then we can apply the **Read-Write** rule. Otherwise, we can apply the **Read2Write** rule.
5) The literal is of the form $x \neq read(A, i)$ and A is not a constant. Then we can apply the **Read-Write** rule.
6) The literal is a (dis)equality between indices or values. Then, we can apply the **UF-Dispatch rule**. □

■

Since the premises and the disjunction of the conclusions of every transformation rule are equisatisfiable, the following lemma holds.

Lemma 5: An ASAT problem $M \mid P$ is satisfiable if and only if at least one leaf of a solved $\mathcal{A}$-derivation tree rooted by $M \mid P$ is not $\bot$.

Now we present our main result.

Theorem 1: The extensional theory of arrays can be decided with $\mathcal{A}$.

Proof: By lemmas 3 and 4, we have that, given a ASAT problem $M \mid P$, a solved $\mathcal{A}$-tree derivation can be obtained. Then by Lemma 5, we can decide if the problem is satisfiable. □ ■

Note that we can apply any strategy in the application of the transformation rules.

IV. Integration of the Solver in DPLL(T)

In the previous section, we showed how to check the satisfiability of conjunctions of literals, but SMT deals with arbitrary formulas. For that purpose, in the DPLL(T) approach to SMT a Boolean engine DPLL(X) works in cooperation with a theory solver $Solver_T$. In its most basic version, the engine enumerates all propositional models of the formula and $Solver_T$ checks the models (seen as conjunctions of literals) for consistency over the theory T. As expected, the input formula is declared satisfiable as soon as a T-consistent propositional model is found.

In our implementation of the $Solver_T$ described in this paper, array expressions are stored using a DAG, where the nodes are array constants and the edges are labelled by the index and the value of the corresponding $write$. In this way we can share information and have an efficient way of applying subtitutions of array constants by $write$ expressions. In addition, (dis)equalities between constants are stored in a Union-Find data structure. In what follows, we give some details about usual optimizations on DPLL(T) systems and how they are implemented in our solver.

Incrementality: there is no need to delay consistency checks until a full propositional model has been found. One can check the T-consistency of partial assignments while they are being built, with the aim of detecting T-inconsistencies at an earlier stage. In order to fully exploit this feature, it is interesting to ask $Solver_T$ to be incremental. That is, once an assignment M has been found T-consistent, processing the addition of a literal l should be done faster (in average) than reprocessing the whole assignment $M \cup \{l\}$ from scratch.

For that purpose, for every array (dis)equality literal we have a *watched pair* of indices, one in each side, that is used for the analysis of satisfiability of the literal. If the literal is not in solved form and we know whether these two indices and their associated values are equal or not, and whether they are significant, we move the watched pair to other indices in order to avoid repeated work in future checks.

Splitting on demand: if some of this information is unknown, we allow the solver not to give a conclusive answer, but rather to ask DPLL(X) to split on a certain equality between indices. This refinement, presented in [9] and called *splitting on demand*, allows reusing the case-splitting infrastructure present in DPLL(X) instead of duplicating it inside $Solver_T$. This simplifies the implementation of all splitting rules presented in the previous section.

Theory propagation: if, when checking a non-solved literal, we have all the information about equalities between indices but not about values, we can propagate an equality between values, applying the **Equality values propagation** rule. Similarly, by successive applications of **Read-Write** we can sometimes infer a disequality between values that is propagated by **UF-Dispatch**. If such a (dis)equality already exists in the input formula we can notify it to the DPLL(X) engine. This optimization, introduced in [10], is very effective in reducing the search space.

Backtracking: sometimes an inconsistency can be detected, and then it is beneficial to backtrack to a point where the assignment was still T-consistent, instead of restarting the search. Hence, we need $Solver_T$ to be backtrackable. Our solver annotates some information with timestamps, e.g. the one given by the Union-Find, and some other information is restored using a trail stack.

In addition, $Solver_T$ has to assist DPLL(X) in identifying the backtrack point by providing an *inconsistency explanation*, that is, given a T-inconsistent set of literals M, it has to provide a small subset of M that is also T-inconsistent. As it is well-known, generating short explanations is a determinant factor in the performance of an SMT solver. In our case, an explanation describes basically the conditions of the transformation rule of Figure 1 that has been applied on a given (dis)equality array literal, together with the explanation of why the relevant indexes and values of both sides of the array literal are there. For this reason, when any of the rules introducing new $write$'s is applied, namely the **Read2Write**, the **Write introduction** or the **Disequality witness introduction** rule,

we remember the literal that has generated it. It is crucial to make these explanations for the introduced $write$'s as short as possible.

V. Experiments

In order to evaluate the $Solver_T$ for arrays described in this paper, we implemented it on top of our BARCELOGIC [11] system[2]. We performed experiments on a 2GHz 2GB Intel Core Duo with a time limit of 300 seconds, comparing our implementation with the four systems that competed at SMT-COMP 2007 (the annual SMT competition[3]) in the QF_AUFLIA division, the only one involving quantifier-free formulas with arrays. These systems are: CVC3 1.2 [12], YICES 1.0 and YICES 1.0.10 [6] and Z3 0.1 [7]. We ran all systems on all available benchmarks in SMT-LIB [13], the largest existing library for SMT problems, discarding families of benchmarks consisting only of trivial problems. The remaining benchmark families were:

- *array-benchs* (25 benchs): a variety of verification conditions involving arithmetic and arrays.
- *cvc* (25 benchs): processor verification conditions involving arithmetic and arrays.
- *qlock2* (52 benchs): unbounded version of the queue lock algorithm. All benchmarks result from parameterizing two single problems. They all contain arithmetic and arrays.
- *storecomm* (2030 benchs), *storeinv* (172 benchs) and *swap* (1368 benchs): benchmarks from the paper [14] encoding simple properties about arrays. They do not contain any arithmetic.

As we can see, most benchmarks involve both arrays and arithmetic, hence forcing us to implement some method for combining the $Solver_T$ for arrays with our solvers for arithmetic (both the difference logic one and the one for full linear arithmetic). It is important to note that BARCELOGIC does not implement any sophisticated combination technique. Unlike what it is done in YICES or Z3, where interface equalities are created on the fly and sophisticated techniques are used to reduce their number, BARCELOGIC implements Delayed Theory Combination [15]. This is much simpler but, since we create all interface equalities upfront, it may significantly slow down the search in some cases. The main reason for that is that, since our arithmetic solvers do not admit (dis)equalities, we have to add clauses, for each interface equality $x = y$, expressing that $x = y \leftrightarrow x \leq y \wedge x \geq y$. This problem is specially acute in the *qlock2* family where, in some cases, up to sixty thousand interface equalities had to be created.

Results are presented in Figure 2, where the column labeled *Total* contains the number of seconds needed to process the whole family. We only count the time for the number of instances solved within 300 seconds and, if not all problems could be handled, we write in parenthesis how many instances could be processed. The column *Max* gives the largest time in seconds needed by a single instance.

	YICES 1.0.10		YICES 1.0	
	Total	Max	Total	Max
array_benchs	52	42	69	52
cvc	5	4	4	3
qlock2	49	5	50	6
storecomm	35	0.1	41	0.1
storeinv	1	0.1	1	0.1
swap	970	130	581	60

	Z3 0.1		CVC3 1.2	
	Total	Max	Total	Max
array_benchs	21	8	496 (16)	294
cvc	1	1	114	57
qlock2	114	37	199 (30)	117
storecomm	37	0.1	993	20
storeinv	8	0.3	691 (162)	76
swap	1431	128	13726 (1263)	275

	BARCELOGIC	
	Total	Max
array_benchs	282	162
cvc	59	38
qlock2	652	55
storecomm	48	0.1
storeinv	22	2
swap	275	9

Fig. 2. Experimental results (times are in seconds)

From the table it can be seen that BARCELOGIC can solve any instance in less than 3 minutes and in fact only one benchmark takes more than one minute. Our system is in general much faster than CVC3, but slower than Z3 and YICES. However, apart from the *qlock2* family, where the huge number of interface equalities greatly affects the search space, the difference wrt. Z3 and YICES is similar to the difference we obtain when we run the systems on formulas not involving arrays. Hence, the difference is probably not due to the array solver but rather to other factors such as heuristics and the worse performance of the arithmetic solvers in BARCELOGIC. In fact, for benchmarks containing a big array component, such as the families *storecomm*, *storeinv* and *swap*, our system is comparable, if not better, which shows that our array solver behaves very well in practice.

VI. Conclusions

In this paper we have described a new theory solver for extensional arrays. As far as we know, this is the first accurate description of an array solver integrated in a state-of-the-art SMT solver and, unlike most state-of-the-art solvers, it is not based on a lazy instantiation of the array axioms. Moreover, our solver is very intuitive and easy-to-understand: after performing a careful analysis on which indices are

[2]The system can be downloaded from `http://www.lsi.upc.edu/~oliveras/espai/fmcad.tar.gz`

[3]See `http://www.smtcomp.org`

relevant in each array, the satisfiability of conjunctions of literals becomes an easy task. We have proved soundness, completeness, and termination of our procedure and shown how it can be integrated to work in a DPLL(T) setting. Finally, we have presented experimental results showing that it performs very well in practice. We want to note that this approach smoothly extends to multidimensional arrays by expressing a position $(i_1, \ldots, i_n)$ in an n-dimensional array as $f(i_1, \ldots, i_n)$, where f is an uninterpreted function symbol.

REFERENCES

[1] J. R. Burch and D. L. Dill, "Automatic Verification of Pipelined Microprocessor Control," in *6th International Conference on Computer Aided Verification, CAV'94*, ser. Lecture Notes in Computer Science, D. L. Dill, Ed., vol. 818. Springer, 1994, pp. 68–80.

[2] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *19th International Conference on Computer Aided Verification, CAV'07*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 519–531.

[3] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani, "A Lazy and Layered SMT(BV) Solver for Hard Industrial Verification Problems," in *19th International Conference on Computer Aided Verification, CAV'07*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 547–560.

[4] R. Alur, "Timed Automata," in *11th International Conference on Computer Aided Verification, CAV'99*, ser. Lecture Notes in Computer Science, N. Halbwachs and D. Peled, Eds., vol. 1633. Springer, 1999, pp. 8–22.

[5] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)," *Journal of the ACM, JACM*, vol. 53, no. 6, pp. 937–977, 2006.

[6] B. Dutertre and L. de Moura, " The YICES SMT Solver," Computer Science Laboratory, SRI International, Tech. Rep., 2006, available at `http://yices.csl.sri.com`.

[7] L. de Moura and N. Bjorner, " Z3: An Efficient SMT Solver," Microsoft Research, Redmond, Tech. Rep., 2007, available at `http://research.microsoft.com/projects/z3`.

[8] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt, "A Decision Procedure for an Extensional Theory of Arrays," in *16th Annual IEEE Symposium on Logic in Computer Science, LICS'01*. IEEE Computer Society, 2001, pp. 29–37.

[9] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Splitting on Demand in SAT Modulo Theories," in *13h International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'06*, ser. Lecture Notes in Computer Science, M. Hermann and A. Voronkov, Eds., vol. 4246. Springer, 2006, pp. 512–526.

[10] R. Nieuwenhuis and A. Oliveras, "DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic," in *17th International Conference on Computer Aided Verification, CAV'05*, ser. Lecture Notes in Computer Science, K. Etessami and S. Rajamani, Eds., vol. 3576. Springer, 2005, pp. 321–334.

[11] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "The Barcelogic SMT Solver," in *20th International Conference on Computer Aided Verification, CAV'08*, ser. Lecture Notes in Computer Science, A. Gupta and S. Malik, Eds. Springer, 2008.

[12] C. Barrett and C. Tinelli, "CVC3," in *19th International Conference on Computer Aided Verification, CAV'07*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 298–302.

[13] C. Tinelli and S. Ranise, "SMT-LIB: The Satisfiability Modulo Theories Library," http://www.smtlib.org.

[14] A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz, "New results on rewrite-based satisfiability procedures," *ACM Transactions on Computational Logic, TOCL*, 2008, to appear.

[15] M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani, "Efficient Satisfiability Modulo Theories via Delayed Theory Combination." in *17th International Conference on Computer Aided Verification, CAV'05*, ser. Lecture Notes in Computer Science, K. Etessami and S. Rajamani, Eds., vol. 3576. Springer, 2005, pp. 335–349.