A Theory of Arrays with set and copy Operations*

(Extended Abstract)

Stephan Falke Carsten Sinz Florian Merz
Institute for Theoretical Computer Science
Karlsruhe Institute of Technology (KIT), Germany
{stephan.falke, carsten.sinz, florian.merz}@kit.edu
http://verialg.iti.kit.edu

Abstract

The theory of arrays is widely used in order to model main memory in program analysis, software verification, bounded model checking, symbolic execution, etc. Nonetheless, the basic theory as introduced by McCarthy is not expressive enough for important practical cases, since it only supports array updates at single locations. In programs, memory is often modified using functions such as `memset` or `memcpy`/`memmove`, which modify a user-specified range of locations whose size might not be known statically. In this paper we present an extension of the theory of arrays with set and copy operations which make it possible to reason about such functions. We also discuss further applications of the theory.

1 Introduction

The theory of arrays is widely used in order to model main memory in formal methods such as program analysis, software verification, bounded model checking, or symbolic execution. In the simplest case, main memory is modelled using a one-dimensional array, but the use of the theory of arrays goes beyond such flat memory models. Reasoning about arrays is thus an essential part of systems that are based on the aforementioned methods.

Since the basic theory of arrays is quite simple, it is insufficient (or at least inconvenient to use) in many application cases. While it supports storing and retrieving of data at specific locations, it does not support the functionality provided by `C` library functions such as `memset` or `memcpy`/`memmove` which operate on regions of locations. While these region-based operations can be broken down into operations on single locations in many cases (e.g., a `memcpy` operation of size 10 can be simulated using 10 read and 10 write operations), this approach does not scale if the involved regions are large. Even worse, the sizes of the affected regions might not be statically known, making it more complicated to break the region-based operation into operations on single locations.

These limitations of the theory of arrays also manifest themselves in tools implementing the formal methods mentioned above. In software bounded model checking tools such as `CBMC` [7] and `ESBMC` [8], calls to `memset` and `memcpy`/`memmove` are handled by including an implementation of these methods and unrolling the loop contained in the implementations. Due to this unrolling, `CBMC` and `ESBMC` are incomplete in their treatment of `memset` and `memcpy`/`memmove`.[1] Our own software bounded model checking tool `LLBMC` [20, 22] was equally incomplete since it relied on user-provided implementations of `memset` and `memcpy`/`memmove` until we implemented the approach discussed in this work.

In this paper we present an extension of the theory of arrays with set and copy operations which make it possible to reason about `memset` and `memcpy`/`memmove`. We discuss several

*This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

[1] The situation is similar in symbolic execution tools such as `EXE` [6] or `KLEE` [5].

implementations of satisfiability solvers for this theory in the context of LLBMC and show with an empirical evaluation that these implementations outperform the unrolling based approach as used in CBMC and ESBMC in all but the most trivial cases.

2 The Theory of Arrays

The (non-extensional) theory of arrays ($\mathcal{T}_{\mathcal{A}}$), as introduced by McCarthy in his seminal paper of 1962 [19], is given by the following signature and axioms:

Sorts	ELEMENT :	elements
	INDEX :	indices
	ARRAY :	arrays
Functions	read :	ARRAY × INDEX → ELEMENT
	write :	ARRAY × INDEX × ELEMENT → ARRAY

Objects of sort ARRAY denote arrays, i.e., maps from indices of sort INDEX to elements of sort ELEMENT. The write function is used to store an element in an array. Its counter-part, the read function, is used to retrieve an element from an array.

The semantics of read and write is given be the following read-*over*-write axioms:[2]

$$p = r \quad \Longrightarrow \quad \mathsf{read}(\mathsf{write}(a, p, v), r) = v \tag{1}$$

$$\neg(p = r) \quad \Longrightarrow \quad \mathsf{read}(\mathsf{write}(a, p, v), r) = \mathsf{read}(a, r) \tag{2}$$

These axioms state that storing the value v into an array a at index p and subsequently reading a's value at index q results in the value v if the indices p and q are identical. Otherwise, the write operation does not influence the result of the read operation.

In a simple implementation of a decision procedure for the theory of arrays, the read-over-write axioms could be applied from left to right using the *if-then-else* operator ITE, i.e., a term $\mathsf{read}(\mathsf{write}(a, p, v), q)$ is replaced by $\mathsf{ITE}(p = q, v, \mathsf{read}(a, q))$. After this transformation has been applied exhaustively, only read operations remain. These can then be treated as uninterpreted functions. Alternatively the resulting formula can be further transformed into pure equality logic using Ackermann's construction: for all array variables a, let P_a be the set of all index arguments that occur in a read operation for a. Then, each occurrence of $\mathsf{read}(a, p)$ is replaced by a fresh variable A_p, and consistency constraints of the form $p_1 = p_2 \implies A_{p_1} = A_{p_2}$ for all $p_1, p_2 \in P_a$ are added as constraints to the formula.

Instead of this eager and expensive approach, modern SMT-solvers use abstraction refinement loops applying lazy axiom instantiation or lemma-on-demand techniques, see, e.g., [12, 4], to efficiently support the theory of arrays.

3 The Theory of Arrays with set and copy

In order to extend $\mathcal{T}_{\mathcal{A}}$ with set and copy operations we restrict INDEX to a linear arithmetical theory containing $+$, $-$, $\leq$, and $<$. Similar to [13], this section is based on the assumption that INDEX $= \mathbb{N}$. Furthermore, $+$, $-$, $\leq$, and $<$ are interpreted in the expected way. Necessary changes in order to replace $\mathbb{N}$ by fixed-width bitvectors are discussed in Section 4.

[2] Here and in the following, all variables are implicitly universally quantified. Also, variables a, b range over arrays, variables p, q, r, s range over indices, and the variable v ranges over elements.

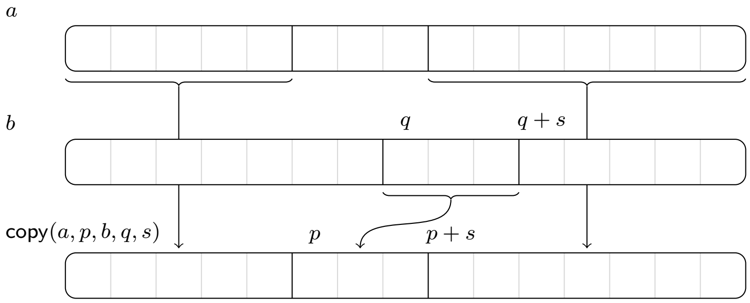

Figure 1: Graphical illustration of copy.

The theory of arrays with set and copy operations ($\mathcal{T}_{\mathcal{ASC}}$) extends $\mathcal{T}_{\mathcal{A}}$ by the following sorts, functions and axioms. The sort SIZE is used for the sizes of set and copy operations. It is introduced as an alias for the sort INDEX since the axioms contain arithmetical operations that combine terms of sort INDEX with terms of sort SIZE.

Sorts	SIZE = INDEX
Functions	set : ARRAY × INDEX × ELEMENT × SIZE → ARRAY set$_\infty$: ARRAY × INDEX × ELEMENT → ARRAY copy : ARRAY × INDEX × ARRAY × INDEX × SIZE → ARRAY copy$_\infty$: ARRAY × INDEX × ARRAY × INDEX → ARRAY

The informal semantics of $\mathsf{set}(a, p, v, s)$ is to set all entries of a that are between p and $p + s - 1$ to the value v. Similarly, $\mathsf{set}_\infty(a, p, v)$ sets all entries from p on upwards to v.

Formally, the semantics of set is given by the following read-*over*-set axioms:

$$p \le r < p + s \implies \mathsf{read}(\mathsf{set}(a, p, v, s), r) = v \tag{3}$$

$$\neg(p \le r < p + s) \implies \mathsf{read}(\mathsf{set}(a, p, v, s), r) = \mathsf{read}(a, r) \tag{4}$$

Similarly, the read-*over*-set$_\infty$ axioms give a semantics to set$_\infty$:

$$p \le r \implies \mathsf{read}(\mathsf{set}_\infty(a, p, v), r) = v \tag{5}$$

$$\neg(p \le r) \implies \mathsf{read}(\mathsf{set}_\infty(a, p, v), r) = \mathsf{read}(a, r) \tag{6}$$

The axioms have some similarity to the read-over-write axioms (1) and (2), but the simple equality comparison in the read-over-write axioms has been replaced by a more complex check for containment in a region of indices.

Informally, copy and copy$_\infty$ are used in order to copy a range of values from one array to another array. This copying process may also involve a shift, i.e., the ranges in the arrays do not need to start at the same index position. See Figure 1 for a graphical illustration of copy.

The semantics of copy is given by the read-*over*-copy axioms

$$p \le r < p + s \implies \mathsf{read}(\mathsf{copy}(a, p, b, q, s), r) = \mathsf{read}(b, q + (r - p)) \tag{7}$$

$$\neg(p \le r < p + s) \implies \mathsf{read}(\mathsf{copy}(a, p, b, q, s), r) = \mathsf{read}(a, r) \tag{8}$$

and the semantics of copy$_\infty$ is specified using the the read-*over*-copy$_\infty$ axioms

$$p \le r \implies \mathsf{read}(\mathsf{copy}_\infty(a, p, b, q), r) = \mathsf{read}(b, q + (r - p)) \tag{9}$$

$$\neg(p \le r) \implies \mathsf{read}(\mathsf{copy}_\infty(a, p, b, q), r) = \mathsf{read}(a, r) \tag{10}$$

Example 1. *Consider the formula*

$$s > 0 \ \wedge \ b = \mathsf{copy}(a, p, a, q, s) \ \wedge \ o < s \ \wedge \ \mathsf{read}(b, p + o) \neq \mathsf{read}(a, q + o)$$

in $\mathcal{T}_{\mathcal{ASC}}$. It is intuitively unsatisfiable since the copy *operation ensures that* $\mathsf{read}(b, p + o)$ *and* $\mathsf{read}(a, q + o)$ *are identical for all $o < s$. The unsatisfiability can formally be shown using reasoning w.r.t. the axioms of $\mathcal{T}_{\mathcal{ASC}}$.*

4 Using Bitvectors as Indices

In principle, the natural numbers used as indices in Section 3 can readily be replaced by fixed-width bitvectors if $+$ and $-$ are replaced by bitvector addition (bvadd) and bitvector subtraction (bvsub) and $\leq$ and $<$ are replaced by unsigned less-than-or-equal (bvule) and less-than (bvult) operations, respectively. In the following discussion, we only consider set and copy.

For the read-over-set axioms (3) this results in the following axioms for the set operation:

$$\begin{aligned} \mathsf{bvule}(p, r) \wedge \mathsf{bvult}(r, \mathsf{bvadd}(p, s)) &\implies \mathsf{read}(\mathsf{set}(a, p, v, s), r) = v \\ \neg(\mathsf{bvule}(p, r) \wedge \mathsf{bvult}(r, \mathsf{bvadd}(p, s))) &\implies \mathsf{read}(\mathsf{set}(a, p, v, s), r) = \mathsf{read}(a, r) \end{aligned}$$

The read-over-copy axioms for copy now read as follows:

$$\begin{aligned} \mathsf{bvule}(p, r) \wedge \mathsf{bvult}(r, \mathsf{bvadd}(p, s)) &\implies \mathsf{read}(\mathsf{copy}(a, p, b, q, s), r) = \mathsf{read}(b, \mathsf{bvadd}(q, \mathsf{bvsub}(r, p))) \\ \neg(\mathsf{bvule}(p, r) \wedge \mathsf{bvult}(r, \mathsf{bvadd}(p, s))) &\implies \mathsf{read}(\mathsf{copy}(a, p, b, q, s), r) = \mathsf{read}(a, r) \end{aligned}$$

Notice that the wrap-around behavior of bitvector arithmetic that is caused by overflows leads to subtle semantic differences compared to the case of natural numbers. There are two cases in which a wrap-around can occur: (a) in the target range of a set or copy operation, i.e., in the computation of $\mathsf{bvadd}(p, s)$, or (b) in the source range of a copy operation, i.e., in the computation of $\mathsf{bvadd}(q, \mathsf{bvsub}(r, p))$.

A target range overflow (a) causes the left-hand side of the implication to be always false for the first axiom and always true for the second axiom for both set and copy. This means that an overflow in the target range of any of the operations turns it into a no-op.

In contrast to target range overflows, source range overflows (b) of a copy operation behave as expected, i.e., the indices from which the values are copied wrap around.[3]

5 Applications

As already mentioned in Section 1, the most immediate use of $\mathcal{T}_{\mathcal{ASC}}$ is to precisely and succinctly model library functions such as `memset` or `memcpy`/`memmove` in C. There are, however, several further applications of the theory for reasoning about programs which we would like to explore in future work:

- Zero-initialization of global variables (as required by the C standard) can be achieved using the set operation.
- Operating systems zero-initialize new memory pages before handing them to a process. This can again be achieved using an appropriate set operation.

[3] If consistency of the two overflow cases is desired, an overflow in the source destination could be ruled out by adding an overflow check $\neg\mathsf{bvuaddo}(q, s)$ to the left-hand side of the implication in the second read-over-copy axiom. This turns the copy operation into a no-op, just like a target overflow does.

- If a memory region should be set to unconstrained values (*havocked*), this can be done using a copy operation from a fresh array variable. Similarly, the same approach could be used in order to model memory-mapped I/O.
- Tracking metadata for memory addresses. For instance, allocation information can be modeled using an array containing information on the allocation state of the locations. Memory allocation and memory de-allocation can then be modelled using the set operation. This makes it possible to develop an alternative to the SMT theory of memory allocation presented in [11] and the memory model presented in [21].

6 Satisfiability Solving

Current SMT-solvers do not support $\mathcal{T}_{\mathcal{ASC}}$ yet. There are, however, several possibilities to reduce the satisfiability problem for quantifier-free $\mathcal{T}_{\mathcal{ASC}}$ formulas to the satisfiability problem for a theory that is supported by existing SMT-solvers. In the following we only consider the set and copy operations, the set_∞ and copy_∞ operations are similar.

Eager Encoding. In this setting, the axioms are applied in a pre-processing step before the resulting formula is passed to an SMT-solver. This pre-processing is done similarly to the case of $\mathcal{T}_{\mathcal{A}}$ as discussed in Section 2 by applying the axioms from left to right using the ITE operator. The main disadvantage of this approach is that the read-over-write axioms have to be applied eagerly as well in the pre-processing in order to eliminate writes that are “between” a read operation and a set or copy operation. The main advantage of the eager encoding is that *any* SMT-solver that can handle $\mathcal{T}_{\mathcal{A}}$ in combination with the index theory can be used.

Example 2. *For the formula from Example 1, the eager encoding produces the formula*

$$s > 0 \ \wedge\ o < s \ \wedge\ \mathsf{ITE}(p < p + o < p + s, \mathsf{read}(a, q + o), \mathsf{read}(a, p + o)) \neq \mathsf{read}(a, q + o)$$

which can easily be seen to be unsatisfiable.

Fully Quantified Axioms. In this approach, set and copy are added as interpreted function symbols whose semantics is given by explicitly stating the quantified axioms. As an example, the fully quantified axiom for set has the form

$$\forall a : \text{ARRAY}, p : \text{INDEX}, v : \text{ELEMENT}, s : \text{SIZE}, r : \text{INDEX}.\ \mathsf{read}(\mathsf{set}(a, p, v, s), r) = \\ \mathsf{ITE}(p \leq r < p + s, v, \mathsf{read}(a, r))$$

The use of quantifiers makes the satisfiability problem hard, but it remains (in principle) decidable if bitvectors are used as indices.

Quantified Axioms Instantiated for Arrays. As in the previous approach, set and copy are added as interpreted function symbols but the fully quantified axiom is partially instantiated for those arrays to which the operators are applied in the formula. Furthermore, each occurrence of, e.g., set is replaced by a fresh interpreted function symbol whose arity coincides with the number of arguments that are still quantified. Then, suitable constraints on this function symbol are added. If the formula contains, e.g., a term of the form $\mathsf{set}(a', p', v', s')$, then the constraint

$$\forall p : \text{INDEX}, v : \text{ELEMENT}, s : \text{SIZE}, r : \text{INDEX}.\ \mathsf{read}(\mathsf{set}_{a'}(p, v, s), r) = \\ \mathsf{ITE}(p \leq r < p + s, v, \mathsf{read}(a', r))$$

is added to the formula.

Quantified Axioms Instantiated for all Arguments. Here, the fully quantified axiom is instantiated even further using all arguments occurring in applications to set and copy. Concretely, if the formula contains a term of the form $\mathsf{set}(a', p', v', s')$, then the following constraint is added to the formula:

$$\forall r : \text{INDEX}.\ \mathsf{read}(\mathsf{set}_{a',p',v',s'}, r) = \mathsf{ITE}(p' \le r < p' + s', v', \mathsf{read}(a', r))$$

Notice that the interpreted function symbol for $\mathsf{set}_{a',p',v',s'}$ is now just a constant symbol.

Instantiation-Based Approach. As in the previous approach, each occurrence of set and copy in the formula is replaced by a fresh constant symbol and constraints on this constant are added. In contrast to the three aforementioned approaches, these constraints do not contain any quantifier. Instead, an instantiation of the quantified constraint from the previous approach is added for all needed read indices. Here, the needed read indices are the (potentially) observed entries of the array produced by the set or copy operation (i.e., entries of the array that are read later on). They are determined by the following algorithm, in which we assume formulas to be represented as terms. Moreover, $\mathrm{root}(x)$ denotes the root symbol of a term x and $y \rhd_{\text{ARRAY}} x$ ($y \unrhd_{\text{ARRAY}} x$) denotes that x is a (non-strict) subterm of y such that the path leading from the root of y to the root of x contains only function symbols of sort ARRAY:

```
/* $x$ is a subterm of the complete formula $\varphi$ with $\mathrm{root}(x) = \mathsf{copy}$ or $\mathrm{root}(x) = \mathsf{set}$ */
function GetNeededReadIndices($\varphi, x$)
    $N = \emptyset$
    /* compute the set of minimal copy superterms of $x$ in $\varphi$ */
    $C = \{y \mid y \rhd_{\text{ARRAY}} x \wedge \mathrm{root}(y) = \mathsf{copy} \wedge \neg\exists z.\ (y \rhd_{\text{ARRAY}} z \rhd_{\text{ARRAY}} x \wedge \mathrm{root}(z) = \mathsf{copy})\}$
    for $c := \mathsf{copy}(a, p, b, q, s) \in C$ do
        if $a \unrhd_{\text{ARRAY}} x$ then
            /* add recursively computed indices */
            $N = N \cup$ GetNeededReadIndices($\varphi, c$)
        end if
        if $b \unrhd_{\text{ARRAY}} x$ then
            /* apply "pointer arithmetic" to the recursively computed indices */
            $N = N \cup \{q + (r - p) \mid p \in$ GetNeededReadIndices($\varphi, c$)$\}$
        end if
    end for
    /* compute the set and write superterms "between" $x$ and $C$ */
    $A = \{y \mid y \unrhd_{\text{ARRAY}} x \wedge \neg\exists z \in C.\ y \rhd_{\text{ARRAY}} z \rhd_{\text{ARRAY}} x\}$
    /* compute the indices used in reads of elements from $A$ */
    $N' = \{y \mid \exists z \in A.\ \mathsf{read}(z, y)$ occurs in $\varphi\}$
    return $N \cup N'$
end function
```

Since the final formula does not contain quantifiers, *any* SMT-solver that can handle $\mathcal{T}_{\mathcal{A}}$ in combination with the index theory can be applied.

Example 3. *For the formula from Example 1, the instantiation-based approach produces the formula*

$$\begin{aligned} s > 0\ \wedge\ & \mathsf{read}(b, p + o) = \mathsf{ITE}(p < p + o < p + s, \mathsf{read}(a, q + o), \mathsf{read}(a, p + o)) \\ & \wedge\ o < s\ \wedge\ \mathsf{read}(b, p + o) \neq \mathsf{read}(a, q + o) \end{aligned}$$

which again can easily be seen to be unsatisfiable. For the construction of the formula, notice that $p + o$ is the only read index that is needed as an instantiation.

7 Evaluation

We have experimented with all possible implementations described in Section 6 for determining the satisfiability of quantifier-free formulas in $\mathcal{T}_{\mathcal{ASC}}$. Since our motivation was the application in our bounded model checking tool LLBMC [20, 22], we have restricted attention to the case where INDEX consists of bitvectors.

The bounded model checking tool LLBMC is a bounded model checker for C and (to some extent) C++ programs. In order to support the complex and intricate syntax and semantics of these programming languages, LLBMC uses the LLVM compiler framework [17] in order to translate C and C++ programs into LLVM's intermediate representation (IR). This IR is then converted into a logical representation and simplified using an extensive set of rewrite rules. The simplified formula is finally passed to an SMT-solver. Distinguishing features of LLBMC in comparison with related tools such as CBMC [7] and ESBMC [8] are its use of a flat, bit-precise memory model, its exhaustive set of built-in checks, and its performance (see [20]).

Within LLBMC, we evaluated the following approaches for $\mathcal{T}_{\mathcal{ASC}}$ satisfiability:

1. The eager encoding and the instantiation-based approach were evaluated in combination with the SMT-solver STP (SVN revision 1625) [12] (alternatively, any other SMT-solver for the logic QF_ABV could be applied, e.g., Boolector [3] or Z3 [9]).

2. The approaches based on fully quantified or partially instantiated axioms have been evaluated in combination with the SMT-solver Z3 (version 3.2) [9], because STP does not support quantifiers.

3. set and copy are simulated using loops on the level of LLVM's IR. Consequently, the boundedness restriction inherent to bounded model checking applies to these loops. This approach was again evaluated in combination with STP.

These approaches have been evaluated on a collection of 55 C and C++ programs (19 programs from LLBMC's test suite that were not specifically written for $\mathcal{T}_{\mathcal{ASC}}$, 19 programs from LLBMC's test suite that were specifically written for $\mathcal{T}_{\mathcal{ASC}}$, and 17 programs from other sources (16 programs from [1] and one program from the KLEE distribution)). Notice that the logical encoding of these programs may contain a set or copy operation for one of several reasons:

- The source code contains an explicit call to memset or memcpy/memmove.

- The compiler may introduce a call to memset in order to initialize global variables.

- Compiler optimizations may detect user-written functionality that essentially constitutes a re-implementation of memset or memcpy/memmove and replace the code by calls to the respective library functions.

- Library-specific implementation details included through header files may result in calls to memset or memcpy/memmove. This is in particular true for C++ programs that use the container classes of the standard template library (STL).

- Default implementations of C++ constructors, especially the copy constructor, may make use of memcpy operations to initialize the memory.

Approach	SMT-solver	Total Time	# Solved Formulas
Instantiation-Based Approach	STP	25.368	55
Eager Encoding	STP	64.008	54
Built-In Loops	STP	225.673	50
Axioms Instantiated for all Arguments	Z3	339.676	45
Axioms Instantiated for Arrays	Z3	420.876	42
Fully Quantified Axioms	Z3	449.767	42

Table 1: Times and success rates for the different approaches.

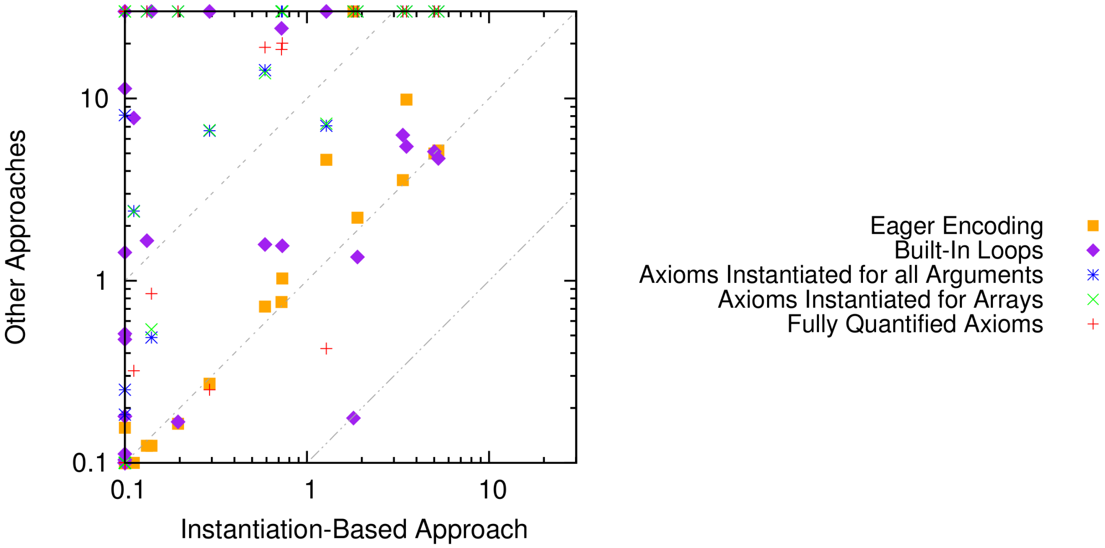

Figure 2: Scatterplot comparing the instantiation-based approach to the other approaches.

For the purpose of this evaluation, also memset and memcpy/memmove with a statically known size were not broken into operations on single locations.

The results of LLBMC on the collection of examples are summarized in Table 1. The reported times are in seconds and only record the time needed for solving the simplified formula, i.e., the time needed for the logical encoding and the simplification of the formula are not included. A timeout of 30 seconds was imposed for each formula and timeouts are contained in the total times. The experiments where performed on an Intel® Core™ 2 Duo 2.4GHz with 4GB RAM.

These results indicate that the instantiation-based approach achieves the best performance, which can also be observed in the scatterplot contained in Figure 2. Notice that Z3 in combination with fully quantified or partially instantiated axioms is significantly slower and less powerful than the approaches that do not involve quantified formulas. Also, the approaches without quantifiers perform better than the naïve implementation using loops on the level of LLVM's IR, where the latter is furthermore incomplete due to the bounded number of loop iterations that can be considered.[4]

[4] This incompleteness does not manifest itself in the evaluation since the number of loop iterations was chosen sufficiently large for each example.

8 Related Work

Decidable extensions of the theory of arrays have been considered before. Suzuki and Jefferson [24] have studied the extension of $\mathcal{T}_\mathcal{A}$ with a restricted use of a permutation predicate. Mateti [18] has described a theory of arrays where entries of an array can be exchanged. Jaffar [16] has investigated reading of array segments but does not discuss writing array segments as needed for set and copy. Ghilardi *et al.* [13] have considered the addition of axioms specifying the dimension of arrays, injectivity of arrays, arrays with domains, array prefixes, array iterators, and sorted arrays. All of these extensions are orthogonal to the set and copy operations considered in this paper. A theory of arrays with constant-valued arrays has been proposed by Stump *et al.* [23]. Our set and set_∞ operators can be seen as proper generalizations of constant-valued arrays. De Moura and Bjørner [10] have introduced *combinatory array logic*, which extends $\mathcal{T}_\mathcal{A}$ by constant-valued arrays and a map combinator.

The satisfiability problem for restricted classes of quantified formulas in the theory of arrays has been investigated as well. The work by Bradley *et al.* [2] identifies the *array property fragment*, where value constraints are restricted by index guards in universally quantified subformulas. Notice that copy and copy_∞ cannot be expressed in the array property fragment due to the "pointer arithmetic" $q + (r - p)$. The set and set_∞ operators, on the other hand, can be defined in the array property fragment. The array property fragment was later extended by Habermehl *et al.* [15, 14], but the "pointer arithmetic" needed for copy and copy_∞ is still not permitted. Finally, Zhou *et al.* [25] have investigated a first-order array theory where the array elements are taken from a finite set.

9 Conclusions

We have presented an extension of the theory of arrays with set and copy operations that can be used in order to model library functions such as C's memset and memcpy/memmove in formal methods such as program analysis, software verification, bounded model checking, or symbolic execution. We have discussed several possible implementations of satisfiability solvers for this theory and have reported on an evaluation in our bounded model checking tool LLBMC [20, 22].

In future work, we are particularly interested in adding "native" support for $\mathcal{T}_{\mathcal{ASC}}$ in SMT-solvers such as STP [12] or Boolector [3]. For this, it is necessary to investigate lazy axiom instantiation or lemma-on-demand techniques for $\mathcal{T}_{\mathcal{ASC}}$ since these techniques have been fundamental for the performance gain that SMT-solvers for $\mathcal{T}_\mathcal{A}$ have experienced in recent years. Furthermore, we will formally prove soundness and completeness of the satisfiability solvers discussed in Section 6, in particular for the instantiation-based approach. Finally, we will investigate the complexity of satisfiability solving for quantifier-free $\mathcal{T}_{\mathcal{ASC}}$ formulas and of the possible solvers discussed in Section 6.

References

[1] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *STTT*, 11(1):69–83, 2009.

[2] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Proc. VMCAI 2006*, volume 3855 of *LNCS*, pages 427–442, 2006.

[3] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proc. TACAS 2009*, volume 5505 of *LNCS*, pages 174–177, 2009.

[4] R. Brummayer and A. Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6:165–201, 2009.

[5] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI 2008*, pages 209–224, 2008.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. *TISSEC*, 12(2):10:1–10:38, 2008.

[7] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS 2004*, volume 2988 of *LNCS*, pages 168–176, 2004.

[8] L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *Proc. ASE 2009*, pages 137–148, 2009.

[9] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS 2008*, volume 4963 of *LNCS*, pages 337–340, 2008.

[10] L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *Proc. FMCAD 2009*, pages 45–52, 2009.

[11] S. Falke, F. Merz, and C. Sinz. A theory of C-style memory allocation. In *Proc. SMT 2011*, pages 71–80, 2011.

[12] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. CAV 2007*, volume 4590 of *LNCS*, pages 519–531, 2007.

[13] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision procedures for extensions of the theory of arrays. *AMAI*, 50(3–4):231–254, 2007.

[14] P. Habermehl, R. Iosif, and T. Vojnar. A logic of singly indexed arrays. In *Proc. LPAR 2008*, volume 5330 of *LNCS*, pages 558–573, 2008.

[15] P. Habermehl, R. Iosif, and T. Vojnar. What else is decidable about integer arrays? In *Proc. FoSSaCS 2008*, volume 4962 of *LNCS*, pages 474–489, 2008.

[16] J. Jaffar. Presburger arithmetic with array segments. *IPL*, 12(2):79–82, 1981.

[17] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO 2004*, pages 75–88, 2004.

[18] P. Mateti. A decision procedure for the correctness of a class of programs. *JACM*, 28:215–232, 1981.

[19] J. McCarthy. Towards a mathematical science of computation. In *Proc. IFIP Congress 1962*, pages 21–28, 1962.

[20] F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *Proc. VSTTE 2012*, volume 7152 of *LNCS*, pages 146–161, 2012.

[21] C. Sinz, S. Falke, and F. Merz. A precise memory model for low-level bounded model checking. In *Proc. SSV 2010*, 2010.

[22] C. Sinz, F. Merz, and S. Falke. LLBMC: A bounded model checker for LLVM's intermediate representation (competition contribution). In *Proc. TACAS 2012*, volume 7214 of *LNCS*, pages 542–544, 2012.

[23] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *Proc. LICS 2001*, pages 29–37, 2001.

[24] N. Suzuki and D. Jefferson. Verification decidability of Presburger array programs. *JACM*, 27:191–205, 1980.

[25] M. Zhou, F. He, B.-Y. Wang, and M. Gu. On array theory of bounded elements. In *Proc. CAV 2010*, volume 6174 of *LNCS*, pages 570–584, 2010.