Formal Methods in Computer-Aided Design 2020

EUFicient Reachability in Software with Arrays

Denis Bueno
Computer Science and Engineering
University of Michigan
Email: dlbueno@umich.edu

Arlen Cox
Institute for Defense Analyses
Center for Computing Sciences
Email: arlen@super.org

Karem Sakallah
Computer Science and Engineering
University of Michigan
Email: karem@umich.edu

***Abstract*—Whether representing strings, heap objects, or numerical vectors, arrays are pervasive in software. Unfortunately, while several software model checkers support arrays, they tend to struggle with many array-manipulating programs due to work expended generating theory lemmas that are ultimately irrelevant or redundant. By judicious abstraction of array operations to the logic of equality with uninterpreted functions (EUF), we show that we can *directly* reason about array reads and *adaptively* learn lemmas about array writes leading to significant performance improvements over existing approaches. We find that our model checker solves more than 100 more SV-COMP benchmarks than SPACER, a leading model checker.**

I. Introduction

Arrays and array-like structures are pervasive in the software world. From C/C++ arrays and vectors to Python lists, it is difficult to find software that doesn't use and manipulate arrays. Despite this, research of software model checkers has largely focused on finding numerical invariants and proving numerical properties of programs. As results of the software verification competition (SV-COMP) show, even when model checkers support arrays, there are a significant number of programs that cannot be automatically verified—some for a lack of expressivity and some for a lack of performance. Our focus is on the latter.

The key challenge that we face is adequately controlling theory reasoning in the SMT solver underlying the model checker. While SMT solvers typically have an array theory and can therefore directly solve array problems, the interface that SMT solvers provide does not provide for adequate incrementality and hinting to enable maximal performance. For instance, we find that, in SV-COMP benchmarks, as many as 90% of the array lemmas that the SMT solver is learning are either redundant or ultimately irrelevant. Most lemmas either do not advance the cause of the model checker or were thrown away by the SMT solver due to imperfect caching. Thus time spent learning those lemmas was wasted effort.

To eliminate this waste, we do incremental inductive model checking on top of an equality with uninterpreted functions (EUF) theory [1]. This removes the need for SMT array theories in the core incremental model checking process, relegating the array theory solely to abstraction refinement operations, and yielding a thousand-fold reduction in the number of operations that do redundant or irrelevant work. Additionally this means that array lemmas are only learned where they are pertinent to proving or disproving the property.

Moreover our strategy addresses a fundamental tension. On the one hand, incremental model checkers [2], which construct a safety proof bit by bit, are particularly scalable because their many individual queries are simple to solve and generalize. On the other hand, these queries lack error path information that could simplify overall checking.

For example, consider model checking the following program, assuming that a, b, and f are distinct constant values:

```
     int[] A; int i, a, b, f;
ℓ1:  A[3] = f;
ℓ2:  A[1] = a;
     A[2] = b;
     assume(1 <= i <= 3);
     if (A[i] == f);
ℓ3:    error();
     else
ℓ4:    exit();
```

The model checker is trying to find if any values of i lead to the error at location ℓ_3. Of course it can reach ℓ_3 if $\mathtt{i} = 3$, which the checker takes two SMT queries to discover. The first query corresponds to reaching ℓ_3, where $\mathtt{A}[\mathtt{i}] = \mathtt{f}$, from ℓ_2. The solver deduces $i \notin \{1, 2\}$, meaning the property may yet be violated, so the checker moves on to the next query, which corresponds to reaching the failure from ℓ_1. The first query involves two array stores and one read; the SMT array theory will generate theory lemmas to deduce that A[i] is not set to f by any assignment from ℓ_2. Several of these lemmas ultimately do not matter, however, since the property is discovered to be violated by the antecedent assignment at ℓ_1.

We study arrays and array abstraction in the context of EUF model checking and make the following contributions:

1) We develop an algorithm for integrating array abstraction into EUFORIA, an EUF-based, incremental, inductive, model checker (Section III).
2) We introduce a refinement procedure for learning relevant array lemmas (Section IV).
3) We evaluate the integration of array abstraction with EUF-based model checking using a variety of device driver benchmarks from SV-COMP (Section V). We find that EUFORIA performs well compared to SPACER and IC3IA.

https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_12

II. Background

a) Equality with Uninterpreted Functions (EUF): We consider a first-order language with equality with signature $\mathcal{S}$ and two common sorts, BOOLS and INTS. Our setting is standard quantifier-free, first-order logic (FOL) with the standard notions of theory, satisfiability, validity, entailment, and models. Much of this background is adapted from previous work [1].

The EUF logic grammar is presented here:

type		production	explanation
term (t)	::=	$\mathsf{x} \mid \mathsf{y} \mid \mathsf{z} \mid \cdots$	0-arity term
	|	$\mathsf{F}(t_1, t_2, \ldots, t_n)$	uninterp. function (UF)
	|	$\mathrm{ite}(f, t_1, t_2)$	if-then-else
atom (a)	::=	$t_1 = t_2$	equality atom
	|	$x \mid y \mid z \mid \cdots$	Boolean atom
	|	$\mathsf{P}(t_1, t_2, \ldots, t_n)$	uninterp. predicate (UP)
formula (f)	::=	a	
	|	$\neg a$	negation
	|	$f_1 \wedge f_2$	conjunction
	|	$f_1 \vee f_2$	disjunction

Atomic formulas (atoms) are made up of Boolean identifiers, uninterpreted predicates (UPs), and equalities between terms. Formulas are made up of terms combined with arbitrary Boolean structure. For simplicity, but without loss of generality, we only consider formulas in negation normal form. A *literal* is a (possibly-negated) atom containing no occurrences of ITE. A *clause* is a disjunction of literals. A *cube* is a conjunction of literals. When convenient, a formula F may be treated as a set of its top-level conjuncts, e.g., $x = 1 \in F$ if $F = (x > 17 \wedge x = 1)$. $a \models b$ means that a entails b. We write uninterpreted objects—terms x, functions F, and predicates P—in sans serif face. The semantics of these formulas is standard.

b) Arrays: We consider a theory of arrays with extensionality and constant-initialized arrays. This theory has the particular function symbols select, store, and const-array. The theory is defined by McCarthy's axioms [3], extended with axioms for extensionality and constant initialization:

$$\forall aije.\ i = j \implies \mathrm{select}(\mathrm{store}(a,i,e),j) = e \tag{1}$$

$$\forall aije.\ i \neq j \implies \mathrm{select}(\mathrm{store}(a,i,e),j) = \mathrm{select}(a,j) \tag{2}$$

$$\forall ab.\ (\forall i.\ \mathrm{select}(a,i) = \mathrm{select}(b,i)) \implies a = b \tag{3}$$

$$\forall ik.\ \mathrm{select}(\text{const-array}(k),i) = k \tag{4}$$

The first two axioms specify array accesses. The third axiom specifies that equal arrays have identical elements at identical indices. The fourth axiom specifies that every index of a constant-initialized array has the initializer value.

We consider this array theory—specifically including equality and constant initialization—because of its utility for software verification. Programs commonly bulk-initialize arrays and array equality allows encodings to be easily composed.

c) Transition Systems for Programs: A *transition system* [4], [5] is a tuple $\mathcal{T} = (X, Y, I, T)$ consisting of a (non-empty) set of *state variables* $X = \{x_1, \ldots, x_n\}$, a (possibly empty) set of *input variables* $Y = \{y_1, \ldots, y_m\}$, and two formulas: I, the *initial states*, and T, the transition relation. Formulas over state variables, or *state formulas*, are identified with the sets of states they denote; for example, the formula $(x_1 = x_2)$ denotes all states where x_1 and x_2 are equal, and other variables may have any value. The *state space* of $\mathcal{T}$ is the set of all valuations to variables in X. The set of *next-state variables* is $X' = \{x'_1, x'_2, \ldots, x'_n\}$. For a formula σ, $\mathrm{Vars}(\sigma)$ denotes the set of state variables free in σ (respectively, $\mathrm{Vars}'(\sigma)$ denotes the set of next-state variables in σ). We may write σ as $\sigma(X)$ when we wish to emphasize that the free variables in σ are drawn solely from the set X, i.e., $\mathrm{Vars}(\sigma(X)) \subseteq X$; similarly for $\sigma(X')$ (also written σ'). The system's *transition relation* $T(X, Y, X')$ is a formula over the current-state, next-state, and input variables.

A (possibly-infinite) sequence of states $\sigma_0(X), \sigma_1(X), \ldots$ is an *execution* of a transition system if $\sigma_0(X) \models I(X)$ and for every pair $(\sigma_i(X), \sigma_{i+1}(X))$, $\sigma_i(X) \wedge T \models \sigma'_{i+1}(X)$.

A *safety property* is specified by a formula, $P(X)$. The *model checking problem* is to determine whether any state satisfying $\neg P(X)$ is reachable through an execution of T. A *counterexample* to a safety property $P(X)$ is a k-step execution such that $\sigma_k(X) \models \neg P(X)$.

A *concrete transition system* (CTS) is defined over bit vector and array state variables and operations in the quantifier-free logic of bit vectors and arrays (`QF_ABV` from SMT-LIB [6]).

III. Model Checking with EUF and Arrays

To better understand how arrays are handled within EUFORIA, we first review EUFORIA's data abstraction approach. It is the inspiration and basis for our array abstraction.

EUFORIA homomorphically maps bit vector operations into uninterpreted functions in order to avoid potentially expensive reasoning (e.g., nonlinear computations). EUF operation abstraction was introduced by Burch and Dill [7] for checking the equivalence between pipelined computer architectures and their single-step specifications. EUFORIA adopts and extends this abstraction to check for general safety properties. For purposes of this paper, we assume there is an abstraction function $[\![.]\!]$ that homomorphically maps a given concrete transition relation to an EUF transition relation. For instance, $[\![x' = x + 1]\!] = (\widehat{x}' = \mathsf{ADD}(\widehat{x}, \widehat{1}))$. State variables, inputs, and constants are mapped to uninterpreted 0-arity terms with hats (e.g., $x \mapsto \widehat{x}$, and $1 \mapsto \widehat{1}$). Operations are mapped to appropriately-named UFs. The crucial property guaranteed by this abstraction is that executions of the EUF transition system over-approximate the executions of the concrete transition system. The details of EUFORIA's translation are available in previous work [1].

EUFORIA performs an incremental induction reachability search based on IC3 [2], a model checking algorithm for finite, Boolean transition systems. EUFORIA uses a counterexample-guided abstraction refinement (CEGAR) [8] approach that extends IC3 to apply to EUF transition systems while retaining termination.

EUFORIA takes a model checking problem as input, (X, Y, I, T, P). It maps the CTS and property to produce a corresponding EUF abstract transition system (ATS) and property, $(\widehat{X}, \widehat{Y}, \widehat{I}, \widehat{T}, \widehat{P})$. EUFORIA then alternates between two phases: EUF reachability and abstraction refinement. EUF reachability searches for a counterexample in the ATS. If no counterexample is found, soundness of the ATS proves that the property holds in the CTS. Otherwise, abstraction refinement analyzes the counterexample to determine if it is feasible in the CTS and, if not, modifies the EUF abstraction to increase its fidelity to the CTS. We first give a brief review of EUF reachability [1] before focusing on refinement.

As in IC3, EUF reachability operates on an iteratively deepened sequence of reachable sets of formulas, R_i, each denoting an over-approximation of the set of states reachable in i transitions ($0 \leq i \leq N$). The algorithm maintains the following invariants:

$$R_0 = \widehat{I}(\widehat{X}) \tag{5}$$

$$R_i \models R_{i+1} \tag{6}$$

$$R_i \models \widehat{P}(\widehat{X}) \qquad (i < N) \tag{7}$$

$$R_{i+1} \text{ over-approximates the image of } R_i \tag{8}$$

EUF reachability computes an inductive invariant for $\widehat{P}$ or a counterexample to the safety property. An *inductive invariant* $\widehat{S}$ for $\widehat{P}$ has the following properties:

$$\widehat{I} \models \widehat{S}, \qquad \widehat{S} \wedge \widehat{T} \models \widehat{S}', \text{ and } \qquad \widehat{S} \models \widehat{P}.$$

This paper brings arrays into the mix. In order to avoid the overhead of instantiating array axioms, array operations and terms may be abstracted. The operations select, store, and const-array are mapped into corresponding uninterpreted functions, select, store, and $\mathsf{const\text{-}array}$ by extending the EUF abstraction mapping $[\![\cdot]\!]$ to array terms and operations as follows:

$$[\![a : \text{Array}]\!] = \widehat{a} \tag{9}$$

$$[\![\text{select}(a, i)]\!] = \mathsf{select}([\![a]\!], [\![i]\!]) \tag{10}$$

$$[\![\text{store}(a, i, x)]\!] = \mathsf{store}([\![a]\!], [\![i]\!], [\![x]\!]) \tag{11}$$

$$[\![\text{const-array}(k)]\!] = \mathsf{const\text{-}array}([\![k]\!]) \tag{12}$$

The array abstraction fits neatly into EUFORIA's data abstraction approach. In fact, this abstraction approach keeps EUFORIA reasoning at the pure (quantifier-free) uninterpreted function level, for which there are efficient decision procedures.

IV. Abstraction Refinement for Arrays

EUF reachability may find an abstract counterexample (ACX). Due to EUF abstraction, the concretized abstract counterexample (CACX) may *not* be a counterexample in the CTS. For example, consider the transition system $\mathcal{E} = (X, Y, I, T)$ defined as

$$(\{A, i\}, \emptyset, [\text{select}(A, i) = 3], [A' = \text{store}(A, i, 3)])$$

with the property, $P = [\text{select}(A, i) = 3]$, which is its own safety invariant. Nevertheless, $[\![P]\!]$ does not hold in $\widehat{\mathcal{E}}$, since EUF abstraction does not preserve the relationship between store and select, and yields the two-step CACX $(I, \text{select}(A, i) \neq 3)$ which is infeasible in the QF_ABV theory. EUFORIA uses this contradictory CACX to *refine*, or increase the fidelity of, the abstraction. Refinement is accomplished by conjoining formulas, called lemmas, to the abstract transition relation.

In this example, EUFORIA learns an instance of McCarthy's axiom (1), to eliminate the spurious behavior caused by the abstraction:

$$\widehat{A}' = \mathsf{store}(\widehat{A}, \widehat{i}, \widehat{3}) \Rightarrow \mathsf{select}(\widehat{A}', \widehat{i}) = \widehat{3}$$

This lemma constrains the abstract state space of $\widehat{\mathcal{E}}$ and is therefore appropriately called a *constraint lemma*. Constraint lemmas restrict the behavior of uninterpreted functions to make them conform more closely to the behavior of their concrete counterparts. A second type of refinement involves learning *expansion lemmas*, which introduce new terms from CACXs. We will discuss these after we present our implementation of abstraction refinement.

A. Implementation of Abstraction Refinement

Our implementation first attempts to derive constraint lemmas by examining individual states and transitions of the abstract counterexample. If none are found, it performs a bounded model check (BMC) of the entire counterexample. If that check is inconsistent, then EUFORIA calculates interpolants from which it derives expansion lemmas. We use a Horn clause solver (SPACER) for convenience to calculate the interpolants; but the interpolants could be obtained using any interpolating theorem prover for QF_ABV. We will discuss each part of refinement in turn.

An n-step *abstract counterexample* is an execution $\widehat{A}_0, \widehat{A}_1, \ldots, \widehat{A}_n$ in $\widehat{T}$ where each $\widehat{A}_i$ ($0 \leq i \leq n$) is a state formula. An abstract formula $\widehat{\sigma}$ is *feasible* if its concretization σ is satisfiable over QF_ABV; therefore, an abstract counterexample is feasible if its concretization is a counterexample in the CTS.

EUFORIA's refinement procedure, BuildCx, is given in Figure 1a; it has three stages. The first stage (lines 1–3) checks whether each $\widehat{A}_i$ is feasible ($0 \leq i \leq n$). The second stage (lines 4–6) checks whether each $\widehat{A}_{i-1} \wedge \widehat{T} \wedge \widehat{A}_i'$ is feasible ($0 < i \leq n$). If an infeasible state or transition is found during the first two stages, we compute an UNSAT core, negate it, and abstract it to form a constraint lemma (in LearnLemma). States and transitions are prioritized over the third stage, BMC, because it is advantageous to learn constraint lemmas, since they make the abstract state space smaller.

Nevertheless, EUFORIA must learn across multiple counterexample steps in general. Therefore, the third stage, BuildBmcCx, performs a BMC query to learn across multiple steps of the counterexample; this is shown in Figure 1b. This stage of refinement has two phases.

a) BuildBmcCx phase one, BMC solving: In phase one (lines 1–2), BmcFormula constructs the instance as

BUILDCX():
Returns true if counterexample is true, false if abstraction is refined
input: counterexample $(\widehat{A}_0, \widehat{A}_1, \ldots, \widehat{A}_n)$ in $\widehat{T}$

if $\exists i \in \{0, \ldots, n\}.\ \neg\mathsf{SAT}[A_i]$ **then**
  LEARNLEMMA(UNSATCORE())
  return false
if $\exists i \in \{1, \ldots, n\}.\ \neg\mathsf{SAT}[A_{i-1} \wedge T \wedge A'_i]$ **then**
  LEARNLEMMA(UNSATCORE())
  return false
return BUILDBMCCX()

(a) The first two stages of refinement: examining concretized states and transitions.

BUILDBMCCX():

1: $\mathcal{B} \leftarrow$ BMCFORMULA()
2: **if** $\neg\mathsf{SAT}[\mathcal{B}]$ **then**
3:   REFINEWITHINTERPOLANTS(UNSATCORE())
4:   **return** false
5: **return** true ▷ feasible counterexample

(b) The third stage of refinement, bounded model checking and interpolant calculation.

Fig. 1: EUFORIA's refinement procedure, BUILDBMCCX.

procedure MBPOUTER(M, f)
  $S \leftarrow \emptyset$; $r \leftarrow$ MBP(f); **return** $S \cup \{\mathrm{Lit}(r)\}$
procedure MBP(f)
  switch f **do**
    case x ▷ x a 0-arity term
      return x
    case $\mathsf{F}(t_1, t_2, \ldots, t_n)$
      return $\mathsf{F}(\mathrm{MBP}(t_1), \mathrm{MBP}(t_2), \ldots, \mathrm{MBP}(t_n))$
    case $\mathrm{ite}(c, t_1, t_2)$ ▷ traverse satisfied branch
      $S \leftarrow S \cup \{\mathrm{Lit}(\mathrm{MBP}(c))\}$
      if $M \models c$ **then return** $\mathrm{MBP}(t_1)$
      else return $\mathrm{MBP}(t_2)$
    case b ▷ b a Boolean variable or its negation
      return $\mathrm{Lit}(b)$
    case $t_1 = t_2$
      return $\mathrm{Lit}(\mathrm{MBP}(t_1) = \mathrm{MBP}(t_2))$
    case $\mathsf{P}(t_1, t_2, \ldots, t_n)$
      return $\mathrm{Lit}(\mathsf{P}(\mathrm{MBP}(t_1), \mathrm{MBP}(t_2), \ldots, \mathrm{MBP}(t_n)))$
    case $f_1 \wedge f_2$
      if $M \models f$ **then return** $\mathrm{MBP}(f_1) \wedge \mathrm{MBP}(f_2)$
      else if $M \models \neg f_1$ **then return** $\mathrm{MBP}(f_1)$
      else return $\mathrm{MBP}(f_2)$ ▷ $M \models \neg f_2$
    case $f_1 \vee f_2$
      if $M \models f_1$ **then return** $\mathrm{MBP}(f_1)$
      else if $M \models f_2$ **then return** $\mathrm{MBP}(f_2)$
      else return MBP $(f_1) \wedge \mathrm{MBP}(f_2)$ ▷ $M \models \neg f$

Fig. 2: Model-based projection of a formula f with model M where $M \models f$. MBPOUTER$(M, f) = S_{\mathrm{MBP}}$ computes a set S_{MBP} of constraints for a formula f such that $M \models S_{\mathrm{MBP}}$ and $S_{\mathrm{MBP}} \models f$. Essentially, it justifies the model of f. In the figure, $\mathrm{Lit}(b) = b$ if $M \models b$ and $\mathrm{Lit}(b) = \neg b$ if $M \models \neg b$.

below by explicitly renaming variables and using multiple copies of T:

$$\begin{aligned}\mathcal{B} = A(X_0) \wedge I(X_0) \wedge T(X_0, Y_1, X_1) \wedge \\ A(X_1) \wedge T(X_1, Y_2, X_2) \wedge \ldots \wedge \\ A(X_{n-1}) \wedge T(X_{n-1}, Y_n, X_n) \wedge A(X_n)\end{aligned}$$

$\mathcal{B}$ is then checked for feasibility. Solving BMC queries is challenging for several reasons. First, there are multiple copies of T. Second, T is monolithic because it encodes the entire program, even though only part of the program is relevant for a given counterexample step. Third, even if we could reduce T at each step by removing irrelevant parts, using a large-step encoding [9] for T means that the reduced T would likely still contain a whole pile of nested Boolean logic, not all of which is necessarily relevant.

At a high level, we address these difficulties by conjoining extra constraints onto $\mathcal{B}$ that significantly prune its search space. These constraints are derived from abstract models gathered during EUFORIA's EUF reachability (see Section III). We use our model-based projection procedure, MBPOuter, given in Figure 2, to derive these extra constraints from the abstract transition relation. We now detail how we solve $\mathcal{B}$.

Let $\widehat{M}_i^{i+1}$ denote the abstract model for the transition $(\widehat{A}_i, \widehat{A}_{i+1})$ in the abstract counterexample $(0 \leq i < n)$. We augment the query $\mathcal{B}$ so that each $T(X_i, Y_{i+1}, X_{i+1})$ is conjoined with the concretization of the constraints in $\mathrm{MBPOuter}(\widehat{M}_i^{i+1}, \widehat{T}(\widehat{X}_i, \widehat{Y}_{i+1}, \widehat{X}_{i+1}))$. The effect of this is that nested logic in $\widehat{T}$ is projected away by justifying the model $\widehat{M}_i^{i+1}$ of the transition. Next, we pre-process $\mathcal{B}$ by an equation solving pass that performs Gaussian elimination and variable elimination.[1] Variables assigned to constants at the top-level will be removed, possibly opening up other elimination opportunities. Linear constraints are solved, leading to further variable elimination. Combining equation-solving with extra constraints addresses difficulties two (T is monolithic) and three (T contains much nested logic). In practice, their combination achieves efficiency far beyond what either does in isolation. Finally, if $\mathcal{B}$ is feasible (BUILDBMCCX line 5), it is a counterexample to the property. If $\mathcal{B}$ is infeasible, BUILDBMCCX enters phase two.

b) BUILDBMCCX *phase two, interpolants:* Phase two is implemented in REFINEWITHINTERPOLANTS, given in Figure 3. BUILDHORN uses $\mathcal{B}$'s UNSAT core to create a (reduced) inductive interpolant sequence problem $\mathcal{B}_{HC}$ [10] using only the constraints from $\mathcal{B}$ that occur in the core. $\mathcal{B}_{HC}$ is a set of recursion-free Horn clauses in which uninterpreted

[1]The `solve-eqs` tactic in Z3.

```
REFINEWITHINTERPOLANTS(core):
  B_HC ← BUILDHORN(core)
  M ← HORNSOLVE(B_HC)
  for i ∈ {1, ..., n} do
      p_i ← GETINTERPOLANT(M, i)
      p_{i+1} ← GETINTERPOLANT(M, i + 1)
      l ← p_{i-1}(X) ∧ body_i(X, Y, X') ∧ ¬p_i(X')
      LEARNLEMMA(l)
```

Fig. 3: Constructs lemmas from an inductive interpolant sequence derived from a solution to (satisfiable) Horn clauses. GETINTERPOLANT$(\mathcal{M}, i)$ returns a formula, the ith interpolant in the interpolant sequence, given a model for $\mathcal{B}_{HC}$.

```
LEARNLEMMA(f):
Precondition: f is unsatisfiable in QF_ABV
1: f̂ ← ABSTRACTANDNORMALIZE(f)          ▷ abstract and eliminate input variables
2: if f contains no inputs then
3:     if VARS(f) ⊆ X then               ▷ only present-state vars
4:         Simplify and add lemma ¬f̂(X̂')
5:     if VARS(f) ⊆ X' then              ▷ only next-state vars
6:         Simplify and add lemma ¬f̂(X̂)
7: Simplify and add lemma ¬f̂
```

Fig. 4: Learns a lemma by abstracting and conjoining $\neg\widehat{f}$ to $\widehat{T}$

predicates p_i stand for step-wise interpolants:

$$
\begin{aligned}
p_0(X_0) &\Leftarrow true \\
p_1(X_1) &\Leftarrow p_0(X_0) \wedge A^*(X_0) \wedge I(X_0) \wedge T^*(X_0, Y_1, X_1) \\
p_2(X_2) &\Leftarrow p_1(X_1) \wedge A^*(X_1) \wedge T^*(X_1, Y_2, X_2) \\
&\vdots \\
p_n(X_n) &\Leftarrow p_{n-1}(X_{n-1}) \wedge A^*(X_{n-1}) \wedge T^*(X_{n-1}, Y_n, X_n) \\
false &\Leftarrow p_n(X_n)
\end{aligned}
$$

where $F^* = \bigwedge\{f \in F \mid f \in \text{UnsatCore}(\mathcal{B})\}$ for $F \in \{A, T\}$.[2] These Horn clauses are satisfiable by construction since $\mathcal{B}$ is infeasible.

For each nontrivial solution to the Horn clauses, we extract a lemma from the corresponding Horn clause as follows:

$$\neg[p_{i-1}(X) \wedge body_i(X, Y, X') \wedge \neg p_i(X')] \qquad 0 < i \leq n \tag{13}$$

where $body_i$ stands for the interpreted body predicates from the rule whose head is p_i.

We now return to the topic of expansion lemmas. Consider a program $x = 3; x = x + 3; assert(x < 7)$. Consider an (infeasible) 2-step counterexample $(x = 3, x \geq 7)$ and its corresponding set of Horn clauses:

$$p_0(3) \tag{14}$$

$$p_1(x') \Leftarrow p_0(x) \wedge x' = x + 3 \tag{15}$$

$$false \Leftarrow p_1(x) \wedge x \geq 7 \tag{16}$$

A solution is $p_0(x) = (x = 3)$ and $p_1(x) = (x = 6)$ which results in the following lemmas (see (13)):

$$\neg[x = 3 \wedge y = x + 3 \wedge y \neq 6] \tag{17}$$

$$\neg[x = 6 \wedge x \geq 7] \tag{18}$$

The key take-away here is that these lemmas introduce the new term 6 into the abstraction, which previously only contained terms from the program text, namely 3, i, 7, and the addition and less-than. These lemmas increase the granularity of the abstraction. This kind of learning is similar to learning new predicates in a predicate abstraction (e.g., [11]).

[2] $\mathcal{B}_{HC}$ could be computed without $\mathcal{B}$'s UNSAT core, but using it promotes learning concise lemmas, because it substantially reduces the complexity of the Horn clause bodies. See equation (13).

Lemmas are expansion lemmas only when the interpolants contain new terms. Using our method implies that the interpolation system itself decides whether a particular lemma is expansive or not; EUFORIA does not make this decision explicitly. EUFORIA's back-end uses SPACER to solve $\mathcal{B}_{HC}$.

Refinement is not guaranteed to succeed. We require quantifier-free interpolants but interpolants for arrays in general are not quantifier-free [12]. Moreover, the interpolant back-end may give up.

To sum up, constraint lemmas specialize UFs to particular concrete behaviors. Expansion lemmas increase the granularity of the EUF abstraction. EUFORIA learns array lemmas only if they crop up in a CACX's contradiction, ensuring that the lemmas are directly relevant to the property that is being checked. Empirically speaking, contradictions usually feature a small handful of UFs which are ultimately relevant to the property, resulting in targeted lemmas. Our process avoids most of the expense of array lemma generation, as we will see in the evaluation.

B. *Exceptionally Lazy Learning of Array Lemmas*

Fundamentally, the procedure LEARNLEMMA (Figure 4) learns its lemmas by negating formulas found to be unsatisfiable in QF_ABV and conjoining them to $\widehat{T}$. It also simplifies the formulas in order to generalize the lemmas as much as possible, specifically by eliminating input variables (line 1). We eliminate input variables from formulas by (1) collecting top-level equalities and computing their equality closure, resulting in equivalence classes of terms; and (2) substituting every input with a member of its equivalence class that doesn't contain inputs (if possible). Next, if the lemma formula is a state formula, then two versions are learned: one on current-state variables and one on next-state variables (lines 2–6).

Consequently, EUFORIA generates property-directed instantiations of array theory axioms. For instance, here is a lemma learned in one of our benchmarks:

$$A \neq \text{const-array}(0) \vee 0 \neq \text{select}(A, i) \tag{19}$$

This lemma is an instance of axiom (4). We also find instances of McCarthy's axiom (1):

$$\text{select}(A', i) = 0 \vee i' \neq i \vee A' \neq \text{store}(A, i', 0) \quad (20)$$

Array lemmas may also include bit-vector function symbols to learn targeted lemmas about composite behavior:

$$B \neq \text{store}(A, i, 0) \vee \text{extract}(7, 0, \text{select}(B, i)) \neq 0 \quad (21)$$

Finally, some lemmas combine multiple array axioms:

$$\text{store}(B, i, 0) \neq A \vee \text{store}(A, i, 0) = A \quad (22)$$

This lemma relates stores and array extensionality. It is not a direct instance of any axiom (1)–(4), but rather a consequence of several instantiations.

We note that LEARNLEMMA is not specialized to produce array lemmas. Rather, it generalizes formulas from unsatisfiable refinement queries that themselves pinpoint which array lemma instantiations to learn. This design allows LEARNLEMMA to produce lemmas that are property-directed combinations of array theory axiom instantiations.

V. EVALUATION

To evaluate EUFORIA, we rely on benchmarks from SV-COMP'17 [13], as they are widely used and relatively well understood. We evaluate on C programs from the Systems_DeviceDriversLinux64_ReachSafety benchmark set, hereafter abbreviated *DeviceDrivers*. This set contains 64-bit C programs and contains "problems that require the analysis of pointer aliases and function pointers." EUFORIA was originally designed for control properties, so our benchmark set includes benchmarks with control properties and arrays.

We consider two other model checkers, SPACER and IC3IA. SPACER [14], [15], [16] is an over- and under-approximation driven incremental model checker that is tightly integrated with Z3. It computes procedure summaries to support checking programs with recursive functions. It is capable of inferring quantified array invariants and uses model-based projection array procedures to lazily instantiate property-directed array axioms, making the checker particularly efficient. IC3IA [11] is an IC3-style CEGAR model checker that implements implicit predicate abstraction. IC3IA's architecture is quite similar EUFORIA's, more similar than SPACER's. As discussed in Cimatti [11], IC3IA is superior to state-of-the-art bit-level IC3 implementations and can support hundreds of predicates, around an order of magnitude more than what explicit predicate abstraction tools practically support. We also evaluated ELDARICA [17], a predicate-abstraction based CEGAR model checker that supports integers, algebraic data types, arrays, and bit vectors. Unfortunately, ELDARICA either threw errors, ran out of time, or ran out of memory on all of our benchmarks, so we do not consider it further.

We use SeaHorn as a front-end to encode programs into Horn clauses. SeaHorn [18] is a verification condition (VC) generator for C and C++ programs that uses LLVM in order to optimize and generate large-step, Horn clause benchmarks in SMT-LIB declare-rel format [19]. Note that we use the term benchmark to refer both to the C programs and their encoded counterparts. Since SeaHorn is not able to produce bit-vector encoded benchmarks, we modified it to produce bit-vector VCs.[3] Moreover, since EUFORIA does not yet support procedure calls, we instruct SeaHorn to inline all procedures, resulting in linear Horn clauses. We ran SeaHorn on each benchmark, limiting it to one hour of runtime and 8GB of memory. SeaHorn can fail to produce a usable benchmark due to lack of resources or because the input is trivially solved during optimization. All told, SeaHorn produced 948 *DeviceDrivers* Horn clause benchmarks out of 2703 original C programs. 687 are safe and 261 are unsafe.

SPACER natively supports Horn clauses, but EUFORIA and IC3IA take VMT files as input. The VMT format [20] is a syntax-compatible extension of the SMT-LIB format that specifies a syntax for labeling formulas denoting initial state, the transition relation, and property. In order to create comparable benchmarks for EUFORIA and IC3IA, we translate the Horn clause benchmarks into VMT using Horn2VMT [21], resulting in 948 VMT files that correspond to the 948 Horn benchmarks. The benchmarks range in size from 2^9 to more than 2^{23}, with a median size of 2^{19}; this size is the number of distinct SMT-LIB expressions used to define (I, T, P). When compressed with gzip, their sizes range from 2K to 153 MB.

All checkers run on 2.6 GHz Intel Sandy Bridge (Xeon E5-2670) machines with 2 sockets, 8 cores with 64GB RAM, running RedHat Enterprise Linux 7. Each checker run was assigned to one socket during execution and was given a 30 minute timeout. For every benchmark solved by any checker, we verified that its result was consistent with other checkers.

A. EUFORIA *compared with* SPACER

Figure 5 shows a scatter plot of runtime for EUFORIA and SPACER on *DeviceDrivers* benchmarks. Overall, EUFORIA solves 491 benchmarks and SPACER solves 386. EUFORIA times out on 33 benchmarks that SPACER solves. SPACER times out on 138 benchmarks that EUFORIA solves.

a) When SPACER *solves* EUFORIA*'s timeouts:* In the 33 cases where spacer was able to solve a benchmark that EUFORIA could not, we identified several causes:

1) SPACER's preprocessor is able to solve 19 benchmarks without even invoking search. By comparison, EUFORIA's front-end takes excessive time to parse and normalize the benchmarks. EUFORIA parses VMT files using MathSAT5, since it the simplest API to do so. In addition to parsing, MathSAT normalizes and simplifies the resulting formula.
2) Another 12 benchmarks are quite large, and the overhead of a monolithic transition relation dominates EUFORIA's abstract reachability. To explain: SeaHorn produces an *explicitly sliced* transition relation which SPACER exploits by making sliced incremental queries. EUFORIA consumes and queries a monolithic transition relation as produced by Horn2VMT.

[3] We worked from SeaHorn commit id 8e51ef84360a602804fce58cc5b7019f1f17d2dc.

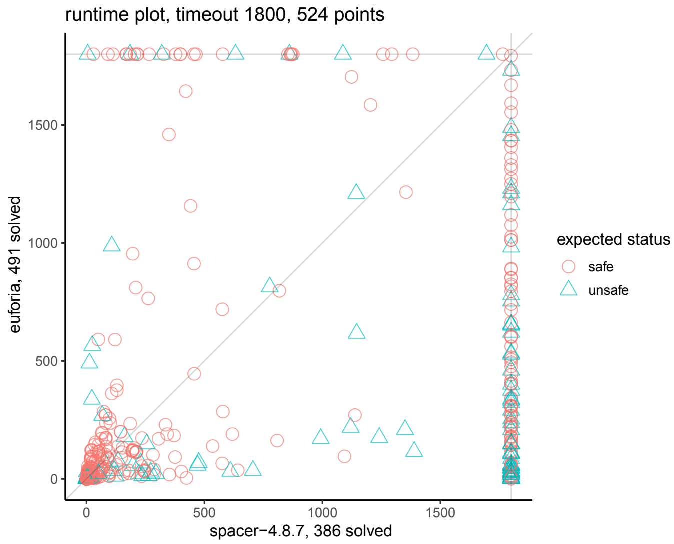

Fig. 5: Benchmarks solved by either solver (or both). Note the points on the right hand side of this plot. Each point is a benchmark that EUFORIA solved within 30 minutes that SPACER did not solve during that time.

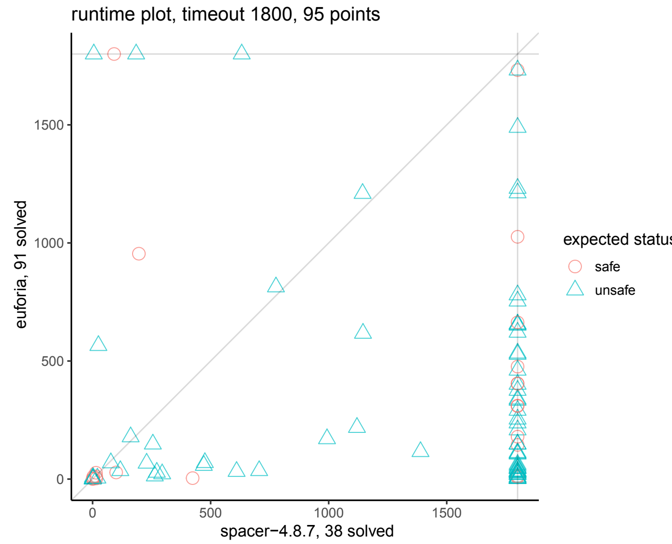

Fig. 6: EUFORIA VS SPACER restricted to those benchmarks that require *at least one* abstraction refinement.

3) In one benchmark EUFORIA gets stuck in a single interpolation query. We suspect this is because some interpolation queries generated by EUFORIA are unexpectedly difficult for SPACER.

In the last un-accounted for benchmark, there was no obvious cause. We believe that front-end improvements would address the issues identified in item 2. For instance, SPACER's preprocessor could be made independent of Z3 so that it could be applied before Horn2VMT.[4] Alternatively, EUFORIA could be integrated into Z3 so that it could exploit the same preprocessing as SPACER, but exploring this remains future work.

b) When EUFORIA *solves* SPACER*'s timeouts:* In the 138 cases where EUFORIA was able to solve a benchmark that SPACER did not, we examined causes. In over half of the cases, SPACER gets stuck solving concrete incremental queries. In the other 52 cases, SPACER gives up before the timeout (it returns unknown). In other words, in every case individual queries were unable to be tackled given the resources constraints. Therefore we emphasize that, in contrast, EUFORIA has the strong benefit of making individual queries predictably fast.

We wondered: is EUFORIA only winning because it hardly needs to do refinement? The answer is no. Figure 6 shows the same scatter plot as Figure 5 but restricted to EUFORIA-solved benchmarks that required at least one abstraction refinement. It shows that EUFORIA requires refinement for many of the benchmarks for which SPACER times out.

[4]We tried dumping the benchmark after SPACER's preprocessing step, but the benchmark was no longer guaranteed to be Horn, so it was not a valid input for encoding to VMT with Horn2VMT.

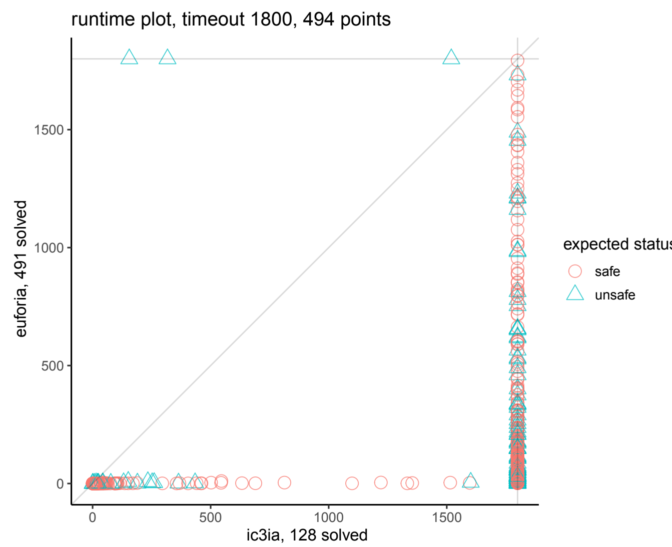

Fig. 7: EUFORIA VS IC3IA

B. EUFORIA *compared with* IC3IA

Figure 7 shows a scatter plot of our results compared with IC3IA. IC3IA solves 128 benchmarks total. Excepting three of these, EUFORIA solves *all* the benchmarks that IC3IA solves, usually in orders of magnitude less time. Our results are significant because IC3IA and EUFORIA are quite similar: both implement a PDR-style [22] algorithm, both operate on *exactly* the same VMT instance encoding, and both are written it C++. They differ in two respects: (1) IC3IA uses (implicit) predicate abstraction and EUFORIA uses EUF abstraction; (2) IC3IA's SMT solver backend is MathSAT5 and EUFORIA's is Z3.

On the benchmarks where EUFORIA times out, two benchmarks get stuck after several seconds in an interpolant query; the other learns a pile of lemmas but doesn't converge in time.

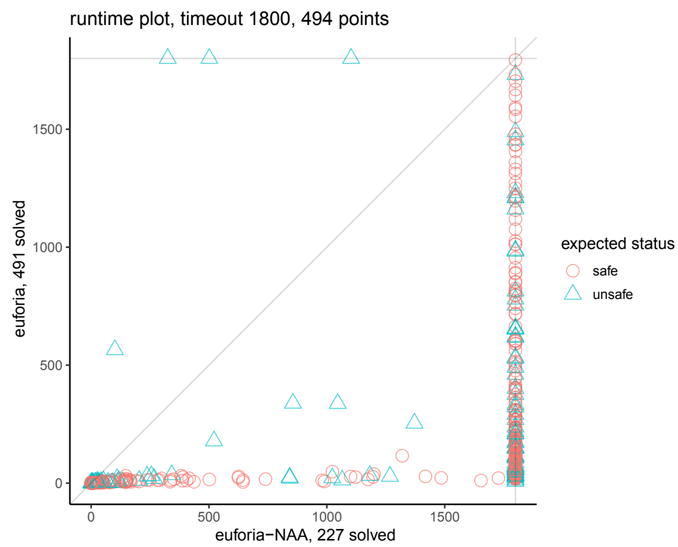

Fig. 8: EUFORIA$_{\text{NAA}}$ (no array abstraction) (x axis) compared with EUFORIA (y axis).

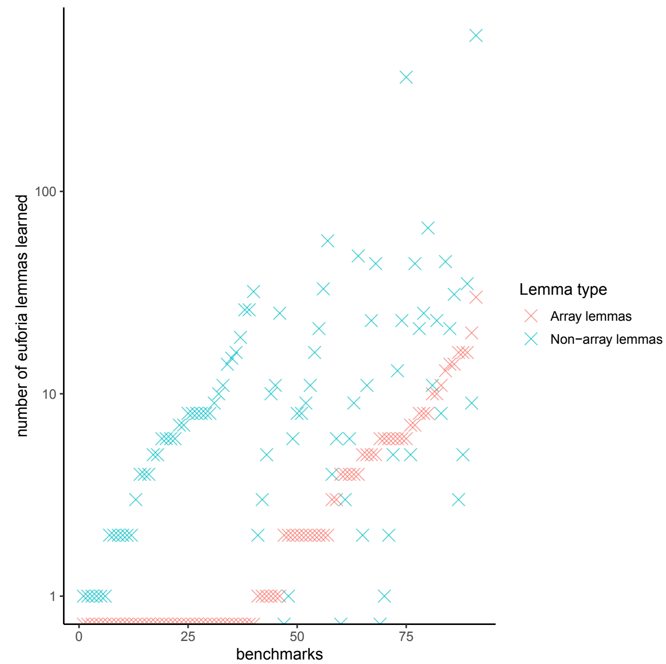

Fig. 9: Breakdown of lemmas as array-related and non array-related on the subset of benchmarks (91) for which any lemma learning was required (y axis is log scale).

C. EUFORIA *and array abstraction*

For solvers that use lazy theory lemma learning or a trigger-based saturation method [23], array lemmas will be learned in response to property-directed queries. Does EUFORIA's array abstraction really provide a benefit over such an approach?

To address this question, we modified EUFORIA to compute a hybrid abstraction using the theory of EUF and arrays. It abstracts bit-vector operations into UFs (as before), but uses array theory operations for arrays. Call this configuration EUFORIA$_{\text{NAA}}$, for No Array Abstraction.

As demonstrated in Figure 8, EUFORIA$_{\text{NAA}}$ is significantly slower almost everywhere and strictly slower in all cases but four. One important difference between EUFORIA and EUFORIA$_{\text{NAA}}$ is an enormous disparity in array theory lemmas learned by the underlying SMT solver. Between configurations, the difference of the number of array theory lemma instantiations is almost two orders of magnitude (1.9), on 95% of the benchmarks; almost four orders of magnitude (3.8), on 50% of the benchmarks; and more than seven orders of magnitude (7.2), on 5%. To calculate this result, we measure the number of array theory axiom instantiations in the underlying SMT solver (Z3). Then, for each benchmark, we took the difference of the logs (base 10) between the two configurations; this quantity is proportional to the order of magnitude difference between the numbers.

We conclude that EUFORIA$_{\text{NAA}}$ spends a lot of time reasoning about arrays despite the fact that EUFORIA required relatively little array reasoning to solve the same benchmarks. Moreover, compared to SPACER's 386 solves, EUFORIA$_{\text{NAA}}$ solves only 227 instances, which (1) shows that array abstraction is critical to performance and (2) gives some additional evidence that SPACER's array projection helps its runtime.

D. EUFORIA *in itself—the role of lemmas*

This section discusses EUFORIA's learned lemmas as detailed in Section IV. Lemmas in general play a relatively minor role; they're only required in 19% of benchmarks that EUFORIA solved (91). Moreover, only 22 benchmarks required interpolants. Figure 9 shows the count of total lemmas learned, broken down by whether EUFORIA learned array lemmas or non-array lemmas. First, we can see that there is a trend that EUFORIA learns fewer array lemmas than data lemmas. Second, all but two benchmarks required fewer than 100 lemmas. These results suggest our benchmarks only depend sparingly on the behavior of memory manipulations, and confirm the suitability of EUFORIA's abstraction. SPACER solves 34 of these benchmarks; out of 34, 14 benchmarks require array lemmas and 20 do not.

VI. Related Work

The relationship between EUF and the theory of arrays has been long recognized [24], [12] and analyzed [25] and exploited in decision procedures [26] and in the implementation of several SMT solvers, including Yices [27] and Z3 [28]. Array terms are compiled into EUF or a ground theory to instantiate the needed array axioms. Our approach lifts EUF outside the SMT solver, to the model checking level, and refines it on demand.

Komuravelli *et al.* introduce a model-based projection for pre-images in order to rewrite array operators into terms in a scalar theory [16]; this algorithm is implemented in SPACER [15] used in our evaluation. Predicate abstraction applies to programs with arrays directly [11], with the limitation that quantifier-free interpolants do not exist in general for the theory of arrays [12]. We inherit that limitation, but contribute a different, inexpensive way to place array constraints in pre-images and refine them lazily.

Broadly, SMT solvers solve constraints over arrays in three ways (sometimes combined): (1) by rewriting selects and stores into a finite number of terms and axiom instantiations in a ground theory, possibly combined with EUF [29], [30], [24], [31], [25], [32], [33], [23], [26]; (2) by abstraction-refinement procedures over the array constraints [34], [35]; (3) by rewriting into (non-abstract) representations which are solved with specialized algorithms [36], [37], [38]. The issue addressed by our paper is applicable to each of these: we use an abstraction that inexpensively supports (limited) array reasoning and we only invoke an SMT array solver at the last possible moment.

VII. Conclusion and Future Work

This paper introduces an approach for model checking software with arrays that avoids substantial computational effort spent in reasoning about arrays by using EUF abstraction. We integrated our approach inside a incremental model checker that natively supports EUF abstraction. Our approach bests stiff competition on control-oriented benchmarks, solving over 100 more benchmarks.

We demonstrated that our approach reduces the amount of redundant or irrelevant array reasoning by several orders of magnitude in most cases. We are eager to investigate the possibilities of expanding our universe of target programs. As software size grows, its sheer size begins to overwhelm the checker, even if the property to prove is relatively simple (for a machine). Inlining all functions only exacerbates the problem. In future work we plan to explore compositional reasoning, in particular analyzing programs with procedures by integrating it efficiently with our EUF abstraction.

We find that for some benchmarks, stronger lemmas are required to speed up convergence. We would like to address this by inferring quantified lemmas during search. One issue is how to generalize counterexamples to quantified lemmas. A second issue is how to keep the abstraction tractable in the presence of quantified lemmas. Both of these issues form important future work.

Acknowledgment

We thank Arie Gurfinkel and Nikolaj Bjørner for their help regarding SPACER and Z3 internals. We thank the anonymous reviewers for their helpful feedback.

References

[1] D. Bueno and K. A. Sakallah, "EUFORIA: Complete software model checking with uninterpreted functions," in *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, ser. Lecture Notes in Computer Science, C. Enea and R. Piskac, Eds., vol. 11388. Springer, 2019, pp. 363–385.

[2] A. R. Bradley, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, R. Jhala and D. A. Schmidt, Eds., vol. 6538, Springer. Springer, 2011, pp. 70–87.

[3] J. McCarthy, "Towards a mathematical science of computation," in *Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962*. North-Holland, 1962, pp. 21–28.

[4] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, 1994.

[5] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction," in *Formal Methods in Computer-Aided Design*. IEEE Computer Society, 2007, pp. 173–180.

[6] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Workshop on Satisfiability Modulo Theories*, A. Gupta and D. Kroening, Eds., 2010.

[7] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, D. L. Dill, Ed., vol. 818. Springer, 1994, pp. 68–80.

[8] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Springer, 2000, pp. 154–169.

[9] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani, "Software model checking via large-block encoding," in *Formal Methods in Computer-Aided Design*. IEEE, 2009, pp. 25–32.

[10] P. Rümmer, H. Hojjat, and V. Kuncak, "Classifying and solving horn clauses for verification," in *Verified Software: Theories, Tools, Experiments*, ser. Lecture Notes in Computer Science, E. Cohen and A. Rybalchenko, Eds., vol. 8164. Springer, 2013, pp. 1–21. [Online]. Available: https://doi.org/10.1007/978-3-642-54108-7_1

[11] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "IC3 modulo theories via implicit predicate abstraction," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, E. Ábrahám and K. Havelund, Eds., vol. 8413. Springer, 2014, pp. 46–61.

[12] D. Kapur, R. Majumdar, and C. G. Zarba, "Interpolation for data structures," in *SIGSOFT FSE*, M. Young and P. T. Devanbu, Eds. ACM, 2006, pp. 105–116. [Online]. Available: https://doi.org/10.1145/1181775.1181789

[13] D. Beyer, "Software verification with validation of results - (report on SV-COMP 2017)," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, A. Legay and T. Margaria, Eds., vol. 10206, 2017, pp. 331–349.

[14] A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke, "Automatic abstraction in smt-based unbounded software model checking," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 846–862. [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8_59

[15] A. Komuravelli, A. Gurfinkel, and S. Chaki, "SMT-Based Model Checking for Recursive Programs," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 17–34. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9

[16] A. Komuravelli, N. Bjørner, A. Gurfinkel, and K. L. McMillan, "Compositional verification of procedural programs using horn clauses over integers and arrays," in *Formal Methods in Computer-Aided Design*, R. Kaivola and T. Wahl, Eds. IEEE, 2015, pp. 89–96.

[17] H. Hojjat and P. Rümmer, "The ELDARICA horn solver," in *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, N. Bjørner and A. Gurfinkel, Eds. IEEE, 2018, pp. 1–7. [Online]. Available: https://doi.org/10.23919/FMCAD.2018.8603013

[18] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The SeaHorn verification framework," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, D. Kroening and C. S. Pasareanu, Eds., vol. 9206. Springer, 2015, pp. 343–361.

[19] N. Bjørner, K. L. McMillan, and A. Rybalchenko, "Program verification as satisfiability modulo theories," in *Workshop on Satisfiability Modulo Theories*, ser. EPiC Series in Computing, P. Fontaine and A. Goel, Eds., vol. 20. EasyChair, 2012, pp. 3–11. [Online]. Available: https://easychair.org/publications/paper/qGkT

[20] F. B. Kessler, *The VMT format*, 2020 (accessed May 13, 2020). [Online]. Available: https://nuxmv.fbk.eu/index.php?n=Languages.VMT
[21] D. Bueno and K. Sakallah, "Horn2VMT: Translating horn reachability into transition systems," in *Workshop on Horn Clauses for Verification and Synthesis*, 2020, p. To appear.
[22] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Formal Methods in Computer-Aided Design*. IEEE, 2011, pp. 125–134.
[23] L. M. de Moura and N. Bjørner, "Generalized, efficient array decision procedures," in *Formal Methods in Computer-Aided Design*. IEEE, 2009, pp. 45–52. [Online]. Available: https://doi.org/10.1109/FMCAD.2009.5351142
[24] D. Kapur and C. G. Zarba, "A reduction approach to decision procedures," University of New Mexico, Tech. Rep., 2005.
[25] A. Goel, S. Krstić, and A. Fuchs, "Deciding array formulas with frugal axiom instantiation," in *International Workshop on Satisfiability Modulo Theories*, ser. SMT '08. New York, NY, USA: ACM, 2008, pp. 12–17. [Online]. Available: http://doi.acm.org/10.1145/1512464.1512468
[26] A. R. Bradley, Z. Manna, and H. B. Sipma, "What's decidable about arrays?" in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, E. A. Emerson and K. S. Namjoshi, Eds., vol. 3855. Springer, 2006, pp. 427–442. [Online]. Available: https://doi.org/10.1007/11609773_28
[27] B. Dutertre and L. M. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, ser. Lecture Notes in Computer Science, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 81–94. [Online]. Available: https://doi.org/10.1007/11817963_11
[28] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.
[29] N. Suzuki and D. Jefferson, "Verification decidability of presburger array programs." CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, Tech. Rep., 1977.
[30] J. Jaffar, "Presburger arithmetic with array segments," *Inf. Process. Lett.*, vol. 12, no. 2, pp. 79–82, 1981. [Online]. Available: https://doi.org/10.1016/0020-0190(81)90007-7
[31] C. Lynch and B. Morawska, "Automatic decidability," in *IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2002, p. 7. [Online]. Available: https://doi.org/10.1109/LICS.2002.1029813
[32] A. Armando, M. P. Bonacina, S. Ranise, and S. Schulz, "New results on rewrite-based satisfiability procedures," *ACM Trans. Comput. Log.*, vol. 10, no. 1, pp. 4:1–4:51, 2009. [Online]. Available: https://doi.org/10.1145/1459010.1459014
[33] J. Christ and J. Hoenicke, "Weakly equivalent arrays," in *International Symposium on Frontiers of Combining Systems*, ser. FroCoS 2015. Berlin, Heidelberg: Springer-Verlag, 2015, pp. 119–134.
[34] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 519–531.
[35] R. Brummayer and A. Biere, "Lemmas on demand for the extensional theory of arrays," *JSAT*, vol. 6, no. 1-3, pp. 165–201, 2009. [Online]. Available: https://satassociation.org/jsat/index.php/jsat/article/view/74
[36] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt, "A decision procedure for an extensional theory of arrays," in *IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2001, pp. 29–37.
[37] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio, "A write-based solver for SAT modulo the theory of arrays," in *Formal Methods in Computer-Aided Design*, A. Cimatti and R. B. Jones, Eds. IEEE, 2008, pp. 1–8. [Online]. Available: https://doi.org/10.1109/FMCAD.2008.ECP.18
[38] P. Habermehl, R. Iosif, and T. Vojnar, "What else is decidable about integer arrays?" in *Foundations of Software Science and Computational Structures*, ser. Lecture Notes in Computer Science, R. M. Amadio, Ed., vol. 4962. Springer, 2008, pp. 474–489.
[39] *Formal Methods in Computer-Aided Design*. IEEE, 2009.