Compositional Verification of Procedural Programs using Horn Clauses over Integers and Arrays

Anvesh Komuravelli
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA

Nikolaj Bjørner
Microsoft Research
Redmond, WA, USA

Arie Gurfinkel
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA

Kenneth L. McMillan
Microsoft Research
Redmond, WA, USA

Abstract—We present a compositional SMT-based algorithm for safety of procedural C programs that takes the heap into consideration as well. Existing SMT-based approaches are either largely restricted to handling linear arithmetic operations and properties, or are non-compositional. We use Constrained Horn Clauses (CHCs) to represent the verification conditions where the memory operations are modeled using the extensional theory of arrays (ARR). First, we describe an exponential time quantifier elimination (QE) algorithm for ARR which can introduce new quantifiers of the index and value sorts. Second, we adapt the QE algorithm to efficiently obtain under-approximations using models, resulting in a polynomial time Model Based Projection (MBP) algorithm. Third, we integrate the MBP algorithm into the framework of compositional reasoning of procedural programs using may and must summaries recently proposed by us. Our solutions to the CHCs are currently restricted to quantifier-free formulas. Finally, we describe our practical experience over SV-COMP'15 benchmarks using an implementation in the tool SPACER.

I. INTRODUCTION

Under-approximating a projection (i.e., existential quantification), for example in computing an image, is a key aspect of many techniques of symbolic model checking. A typical (though not ubiquitous) approach to this is what we call *Model-based Projection* (MBP) [17]: we generalize a particular point in the space of the image (obtained using a model) to a subset of the image that contains it. In some cases, the purpose is to compute the exact image by a series of under-approximations [12]. In other cases, such as IC3 [6], the purpose of MBP is to produce a relevant proof sub-goal. When the number of possible generalizations is finite, we say that we have a *finite MBP* which allows us to compute the exact image by iterative sampling, or to guarantee that the branching in our proof search is finite.

The feasibility of a finite MBP depends on the underlying logical theory. Finite MBPs exist for propositional logic [12], [16] and Linear Integer Arithmetic (LIA) with a divisibility predicate [17], and have been applied in both hardware and software model checking. LIA is often adequate for software verification, provided that heap and array accesses can be eliminated. This can be done by abstraction, or by inlining all procedures and performing compiler optimizations to lower

memory into registers (e.g., [2], [15]). However, the inlining approach has many drawbacks. It can expand the program size exponentially, it cannot handle recursion, and it is not always feasible to eliminate heap and array accesses.

We address this issue here by considering the problem of MBP for the extensional theory of arrays (ARR). We find that a finite MBP exists that can be computed in polynomial time when only array-valued variables are projected. Projecting variables of index and value sorts is not always possible, since the quantifier-free fragments of the theory combinations are not guaranteed to be closed under projection. We therefore take a pragmatic approach to MBP that may not always converge to the exact projection. This allows us to handle, for example, the combination of ARR and LIA.

We test the effectiveness of this approach using the model checking framework of SPACER [17]. This SMT-based framework makes use of MBP to produce proof sub-goals for Hoare-style procedure-modular proofs of recursive programs. The ability to reason with ARR makes it possible to handle heap-allocating programs without inlining procedures, as the heap can be faithfully modeled using ARR [14]. This leads to significant improvements in scalability, when compared to the use of LIA alone with inlining, as measured using benchmark programs from the 2015 Software Verification Competition (SVCOMP 2015) [4]. Not inlining the programs also has the advantage that we generate procedure-modular proofs (containing procedure summaries) that might be reusable in various ways (e.g., [11]).

In summary, we (a) describe an exponential rewriting procedure for projecting array variables (Sec. III-A), (b) adapt this procedure to obtain a polynomial-time (per model) finite MBP for projecting array variables (Sec. III-B), (c) integrate this with existing MBP procedures for Linear Arithmetic (Sec. III-C) in the SPACER framework obtaining a new compositional proof search algorithm (Sec. IV), and (d) evaluate the algorithm experimentally using SVCOMP benchmarks (Sec. V).

II. PRELIMINARIES

We consider a first-order language with equality whose signature $\mathcal{S}$ contains basic sorts (e.g., `bool` of Booleans, `int` of integers, etc.) and array sorts. An array sort $\mathrm{arr}(I, V)$ is parameterized by a sort of indices I and a sort of values V.

We assume that I is always a basic sort. For every array sort $\text{arr}(I,V)$, the language has the usual function symbols $rd : \text{arr}(I,V) \times I \to V$ and $wr : \text{arr}(I,V) \times I \times V \to \text{arr}(I,V)$ for reading from and writing to the array. Intuitively, $rd(a,i)$ denotes the value stored in the array a at the index i and $wr(a,i,v)$ denotes the array obtained from a by replacing the value at the index i by v. We use the following axioms for the extensional theory of arrays (ARR):

Read-after-write
$\forall a : \text{arr}(I,V) \; \forall i,j : I \; \forall v : V$

$$(i = j \implies rd(wr(a,i,v),j) = v) \wedge$$
$$(i \neq j \implies rd(wr(a,i,v),j) = rd(a,j))$$

Extensionality
$\forall a,b : \text{arr}(I,V) \cdot (\forall i : I \cdot rd(a,i) = rd(b,i)) \implies a = b$

Intuitively, the first schema says that after modifying an array a at index i, a read results in the new value at index i and $rd(a,j)$ at every other index j. The second schema says that if two arrays agree on the values at every index location, the arrays are equal. We use an over-bar to denote a vector. We write $\overline{x} : S$ to denote that every term in vector $\overline{x}$ has sort S, $\overline{x}(k)$ to denote the kth component of $\overline{x}$, and $y \in \overline{x}$ to denote that y is equal to some component of $\overline{x}$, i.e., $\bigvee_{k=1}^{|\overline{x}|} y = \overline{x}(k)$. Let $\overline{i} : I$ and $\overline{v} : V$ be vectors of index and value terms of the same length m. We write $wr(a,\overline{i},\overline{v})$ to denote $wr(wr(\ldots wr(a,\overline{i}(0),\overline{v}(0))\ldots),\overline{i}(m),\overline{v}(m))$. Unless specified otherwise, $\mathcal{S}$ contains no other symbols.

For arrays a and b of sort $\text{arr}(I,V)$, and a (possibly empty) vector of index terms $\overline{i}$, we write $a =_{\overline{i}} b$ to denote $\forall j : I \cdot (j \notin \overline{i} \implies rd(a,j) = rd(b,j))$ and call such formulas *partial equalities* [20]. Using extensionality, one can easily show the following

$$a =_{\emptyset} b \equiv a = b \tag{1}$$

$$wr(a,j,v) =_{\overline{i}} b \equiv \begin{array}{l}\left(j \in \overline{i} \wedge a =_{\overline{i}} b\right) \vee \\ \left(j \notin \overline{i} \wedge a =_{\overline{i},j} b \wedge rd(b,j) = v\right)\end{array} \tag{2}$$

$$a =_{\overline{i}} b \equiv \exists \overline{v} : V \cdot a = wr(b,\overline{i},\overline{v}) \tag{3}$$

We write $\varphi(\overline{x})$ for a formula φ with free variables $\overline{x}$, and we treat ϕ as a predicate over $\overline{x}$. We also write $\varphi[t]$ to to indicate that a term or formula t occurs in φ at some syntactic position.

Given formulas $\varphi_A(\overline{x},\overline{z})$ and $\varphi_B(\overline{y},\overline{z})$ with $\overline{x} \cap \overline{y} = \emptyset$ and $\varphi_A \implies \varphi_B$, a Craig Interpolant [7], denoted $\text{ITP}(\varphi_A, \varphi_B)$, is a formula $\varphi_I(\overline{z})$ such that $\varphi_A \implies \varphi_I$ and $\varphi_I \implies \varphi_B$.

III. QE AND MBP FOR THE THEORY ARR

By projection of a variable we mean elimination of an existential quantifier. Consider a formula φ of the form $\exists \overline{x} \cdot \varphi_{qf}(\overline{x},\overline{y})$ where φ_{qf} is quantifier-free. The problem of *quantifier elimination* (QE) in φ is to find a logically equivalent quantifier-free formula $\psi(\overline{y})$. In this case, we say that ψ is the result of projecting $\overline{x}$ in φ_{qf}.

A *model-based projection* (MBP) for φ is an operator $Proj$ that takes a model M of φ_{qf} and returns a quantifier-free formula $\psi_M(\overline{y})$ such that $M \models \psi_M$ and ψ_M entails φ. The

operator $Proj$ is a *finite MBP* if its image is finite up to logical equivalence (that is, over all models we obtain only finitely many semantically distinct formulas).[1] In this case, we obtain the exact projection as the disjunction of the image of $Proj$. We will refer to $Proj(M)$ as a *generalization* of M.

In some cases, there is a trivial approach to MBP that we will call the *substitution approach*. We simply substitute for each variable x in φ a constant that is equal to x in the given model M (for example, a numeric literal). This approach was taken for propositional logic by Ganai et al. [12]. For theories that admit models of unbounded size (e.g., LIA), however, this does not yield a finite MBP, as the number of distinct generalizations we obtain can be infinite.

Instead, we can take the approach used for Linear Real Arithmetic and LIA in our earlier work [17]. Suppose that for the given theory we have a QE procedure that produces a formula with an exponential (or higher) number of disjunctions. We can adapt this procedure to an MBP by always choosing just one disjunct that is true in the given model M. The result may be a procedure that is polynomial for any given model, though the number of distinct generalizations is exponential. We will show how to apply this idea for the projection of array-valued variables in the theory of arrays ARR. When combining this theory with LIA, we will find that some variables of index and value sorts must be eliminated by the substitution method, which gives us a useful MBP but not necessarily a *finite* MBP.

A. Quantifier elimination for ARR

Consider an existentially quantified formula $\exists a : \text{arr}(I,V) \cdot \varphi$ where φ is quantifier-free. We restrict our discussion to infinite interpretations of I. While we cannot always obtain an equivalent quantifier-free formula, our objective here is to obtain an equivalent existentially quantified formula where every quantifier (if any) is of the sort V.

ARRAYQE($\exists a \cdot \varphi$)
1    $\varphi_1 \leftarrow (\text{ELIMWR}^*)(\exists a \cdot \varphi)$
2    $\varphi_2 \leftarrow (\text{CASESPLITEQ}^*; \text{FACTORRD}^*)(\varphi_1)$
3    $\left(\bigvee_{k=1}^{n} \delta_k\right) \leftarrow \text{LIFTEQDISEQRD}(\varphi_2)$
4   **for** $k \in [1,n]$ **do**
5       $\psi_k \leftarrow (\text{ELIMEQ}; \text{ELIMDISEQ}; \text{ACKERMANN})(\delta_k)$
6   **return** $\bigvee_{k=1}^{n} \psi_k$

Algorithm 1: QE for $\exists a \cdot \varphi$, where a is an array variable.

Our algorithm is inspired by the decision procedure for the quantifier-free fragment of ARR by Stump *et al.* [20]. At a high level, the QE algorithm proceeds in 3 steps: (i) eliminate write terms using the read-after-write axiom schema and partial equalities over arrays, (ii) eliminate (partial) equalities and disequalities over arrays, and (iii) eliminate read terms over arrays. Alg. 1 shows the pseudo-code for our QE algorithm ARRAYQE using the rewrite rules in Fig. 1, 2, and 3. Each rule rewrites the formula above the line to the logically equivalent formula below the line. We use regular expression notation

---

[1] MBP as defined in [17] corresponds to *finite MBP* here.

$$\text{ELIMWRRD} \; \frac{\varphi[rd(wr(t,i,v),j)]}{(i=j \wedge \varphi[v]) \vee (i \neq j \wedge \varphi[rd(t,j)])}$$

$$\text{ELIMWREQ} \; \frac{\varphi[wr(t_1,j,v) =_{\bar{i}} t_2]}{\left(j \in \bar{i} \wedge \varphi[t_1 =_{\bar{i}} t_2]\right) \vee \left(j \notin \bar{i} \wedge \varphi[t_1 =_{\bar{i},j} t_2 \wedge v = rd(t_2,j)]\right)}$$

$$\text{PARTIALEQ} \; \frac{\varphi[t_1 = t_2]}{\varphi[t_1 =_\emptyset t_2]} \; t_i\text{'s have array sort} \qquad \text{TRIVEQ} \; \frac{\varphi[t =_{\bar{i}} t]}{\varphi[\top]} \qquad \text{SYMM} \; \frac{\varphi[t_1 =_{\bar{i}} t_2]}{\varphi[t_2 =_{\bar{i}} t_1]} \; \begin{array}{l} t_2 \text{ is a write term} \\ \text{but } t_1 \text{ is not} \end{array}$$

$$\text{ELIMWR} = (\text{ELIMWRRD} \mid \text{ELIMWREQ} \mid \text{PARTIALEQ} \mid \text{TRIVEQ} \mid \text{SYMM})$$

Fig. 1: Rewriting rules to eliminate write terms. ELIMWR denotes one of the rules chosen non-deterministically.

$$\text{CASESPLITEQ} \; \frac{\exists a \cdot \varphi[a =_{\bar{i}} t]}{\exists a \cdot ((a =_{\bar{i}} t \wedge \varphi[\top]) \vee (\neg(a =_{\bar{i}} t) \wedge \varphi[\bot]))} \qquad \text{FACTORRD} \; \frac{\exists a \cdot \varphi[rd(a,t)]}{\exists a, s \cdot (\varphi[s] \wedge s = rd(a,t))} \; \begin{array}{l} s \text{ is fresh, } t \text{ does not} \\ \text{contain array terms} \end{array}$$

Fig. 2: Rewriting rules to factor out equalities and read terms on the quantified array variable.

$$\text{ELIMEQ} \; \frac{\exists a \cdot (a =_{\bar{i}} t \wedge \varphi)}{\exists \overline{v} \cdot \varphi[wr(t,\bar{i},\overline{v})/a]}$$

where a does not appear in t and $\overline{v}$ denotes fresh variables

$$\text{ELIMDISEQ} \; \frac{\exists a \cdot \left(\varphi \wedge \bigwedge_{k=1}^{m} \neg(a =_{\bar{i}_k} t_k)\right)}{\exists a \cdot \varphi}$$

where $m \in \mathbb{N}$, a does not appear in any t_k, and a appears in φ only in read terms over a

$$\text{ACKERMANN} \; \frac{\exists a \cdot \left(\varphi \wedge \bigwedge_{k=1}^{m} s_k = rd(a,t_k)\right)}{\varphi \wedge \bigwedge_{1 \leq k < \ell \leq m} (t_k = t_\ell \implies s_k = s_\ell)}$$

where $m \in \mathbb{N}$ and a does not appear in φ, s_k's, or t_k's

Fig. 3: Rewriting rules for QE of arrays.

to express sequences of rewrites. In particular, Kleene star applied to a rule denotes the rule's application to a fixed point.

Line 1 of ARRAYQE eliminates write terms using the rewrite rules in Fig. 1. Here ELIMWR denotes a rule in Fig. 1 chosen non-deterministically. ELIMWRRD rewrites terms using the read-after-write axiom and ELIMWREQ rewrites partial equalities using Eq. (2). PARTIALEQ converts equalities into partial equalities using Eq. (1). TRIVEQ eliminates trivial partial equalities with identical arguments and SYMM ensures that write terms on the r.h.s. of equalities are also eliminated.

Line 2 of ARRAYQE rewrites the formula by case-splitting on partial equalities on the array quantifier a (via CASES-

PLITEQ) followed by factoring out read terms over a by introducing new quantifiers of sort V (via FACTORRD). Note that, as presented, these two rules are not terminating as the partial equalities and read terms are preserved in the conclusion of the rules. However, one can easily ensure that a given partial equality or read term is considered exactly once by first computing the set of all partial equalities and read terms in the formula and processing them in a sequential order. The details are straightforward and are left to the reader.

LIFTEQDISEQRD on line 3 of ARRAYQE performs Boolean rewriting and returns an equivalent disjunction such that in every disjunct, the partial equalities, array disequalities, and equalities over read terms appear at the end as conjuncts, in that order. For each disjunct, line 5 applies the rules in Fig. 3 to eliminate the array quantifier a. ELIMEQ obtains a substitution term for a using the equivalence in Eq. (3). ELIMDISEQ is applicable when the disjunct contains no partial equalities and given that the domain of interpretation of I is infinite, one can always satisfy the disequalities and hence, they can simply be dropped. ACKERMANN performs the Ackermann reduction [1] to eliminate the read terms.

Note that while the rewrite rules are applicable to all array terms and equalities in the original formula, in practice, we only need to apply them to eliminate the relevant terms containing the array quantifier a. See Fig. 4 for an illustration of ARRAYQE on an example.

Correctness and Complexity. We can show the following properties of ARRAYQE.

Theorem 1: $\text{ARRAYQE}(\exists a : \text{arr}(I,V) \cdot \varphi)$ returns $\exists \overline{v} : V \cdot \rho$, where ρ is quantifier-free and $\exists \overline{v} \cdot \rho \equiv \exists a \cdot \varphi$.

Theorem 2: $\text{ARRAYQE}(\exists a \cdot \varphi)$ terminates in time exponential in the size of φ.

$$\exists a \cdot (b = wr(a, i_1, v_1) \lor (rd(wr(a, i_2, v_2), i_3) > 5 \land rd(a, i_4) > 0))$$

$$\equiv \exists a \cdot \begin{matrix} (\boldsymbol{i_2 = i_3} \land (b = wr(a, i_1, v_1) \lor (\boldsymbol{v_2 > 5} \land rd(a, i_4) > 0))) \lor \\ (\boldsymbol{i_2 \neq i_3} \land (b = wr(a, i_1, v_1) \lor (\boldsymbol{rd(a, i_3) > 5} \land rd(a, i_4) > 0))) \end{matrix} \qquad \{\textsc{ElimWrRd}\}$$

$$\equiv \exists a \cdot \begin{matrix} (i_2 = i_3 \land ((\boldsymbol{a =_{i_1} b} \land \boldsymbol{rd(b, i_1) = v_1}) \lor (v_2 > 5 \land rd(a, i_4) > 0))) \lor \\ (i_2 \neq i_3 \land ((\boldsymbol{a =_{i_1} b} \land \boldsymbol{rd(b, i_1) = v_1}) \lor (rd(a, i_3) > 5 \land rd(a, i_4) > 0))) \end{matrix} \qquad \{\textsc{PartialEq}; \textsc{ElimWrEq}\}$$

$$\equiv \exists a \cdot \left(\begin{matrix} \left(\boldsymbol{a =_{i_1} b} \land \begin{matrix} (i_2 = i_3 \land (rd(b, i_1) = v_1 \lor (v_2 > 5 \land rd(a, i_4) > 0))) \lor \\ (i_2 \neq i_3 \land (rd(b, i_1) = v_1 \lor (rd(a, i_3) > 5 \land rd(a, i_4) > 0))) \end{matrix} \right) \lor \\ \left(\boldsymbol{\neg(a =_{i_1} b)} \land \begin{matrix} (i_2 = i_3 \land (v_2 > 5 \land rd(a, i_4) > 0)) \lor \\ (i_2 \neq i_3 \land (rd(a, i_3) > 5 \land rd(a, i_4) > 0)) \end{matrix} \right) \end{matrix} \right) \qquad \{\textsc{CaseSplitEq}\}$$

$$\equiv \exists a, s_3, s_4 \cdot \left(\begin{matrix} \left(a =_{i_1} b \land \underbrace{\begin{matrix} (i_2 = i_3 \land (rd(b, i_1) = v_1 \lor (v_2 > 5 \land \boldsymbol{s_4} > 0))) \lor \\ (i_2 \neq i_3 \land (rd(b, i_1) = v_1 \lor (\boldsymbol{s_3} > 5 \land \boldsymbol{s_4} > 0))) \end{matrix}}_{\varphi_1} \right) \lor \\ \left(\neg(a =_{i_1} b) \land \underbrace{\begin{matrix} (i_2 = i_3 \land (v_2 > 5 \land \boldsymbol{s_4} > 0)) \lor \\ (i_2 \neq i_3 \land (\boldsymbol{s_3} > 5 \land \boldsymbol{s_4} > 0)) \end{matrix}}_{\varphi_2} \right) \\ \land \boldsymbol{s_3 = rd(a, i_3)} \land \boldsymbol{s_4 = rd(a, i_4)} \end{matrix} \right) \qquad \{\textsc{FactorRd}\}$$

$$\equiv \exists a, s_3, s_4 \cdot \begin{matrix} (\varphi_1 \land \boldsymbol{a =_{i_1} b} \land \boldsymbol{s_3 = rd(a, i_3)} \land \boldsymbol{s_4 = rd(a, i_4)}) \lor \\ (\varphi_2 \land \boldsymbol{\neg(a =_{i_1} b)} \land \boldsymbol{s_3 = rd(a, i_3)} \land \boldsymbol{s_4 = rd(a, i_4)}) \end{matrix} \qquad \{\textsc{LiftEqDiseqRd}\}$$

$$\equiv \exists v, s_3, s_4 \cdot (\varphi_1 \land s_3 = rd(a, i_3) \land s_4 = rd(a, i_4)) \, [\boldsymbol{wr(b, i_1, v)/a}] \lor \qquad \{\textsc{ElimEq}\}$$
$$\exists a, s_3, s_4 \cdot (\varphi_2 \land s_3 = rd(a, i_3) \land s_4 = rd(a, i_4)) \qquad \{\textsc{ElimDiseq}\}$$

$$\equiv \exists v, s_3, s_4 \cdot (\varphi_1 \land s_3 = rd(a, i_3) \land s_4 = rd(a, i_4)) \, [wr(b, i_1, v)/a] \lor$$
$$\exists s_3, s_4 \cdot (\varphi_2 \land (\boldsymbol{i_3 = i_4 \implies s_3 = s_4})) \qquad \{\textsc{Ackermann}\}$$

Fig. 4: Illustrating ARRAYQE on an example.

B. Model Based Projection

In this section, we will assume that for a satisfiable formula we can obtain a finite representation of a model of the formula and that we can effectively evaluate the truth of any formula in this model. This is possible for ARR and its combinations with LIA and propositional logic. The ability to evaluate allows us to strengthen a formula in a way that preserves a given model. Suppose we have a formula $\varphi[\psi_1 \lor \psi_2]$ with model M, where the sub-formula $\psi_1 \lor \psi_2$ occurs positively (under an even number of negations) in φ. If we also have $M \models \psi_1$, then $M \models \varphi[\psi_1]$ and clearly, $\varphi[\psi_1]$ entails φ. This gives us a way to eliminate a disjunction while preserving a given model and maintaining an under-approximation. If neither ψ_1 nor ψ_2 is true in M, we can similarly replace φ with $\varphi[\bot]$. These transformations are expressed as MBP rewrite rules in Fig. 5.

For each QE rule R, we can produce a corresponding under-approximate rule R_M that preserves model M. This rule can be written $R \, ; \, (\textsc{MbpLeft} \mid \textsc{MbpRight} \mid \textsc{MbpVac})^*$. In practice, we can choose to only apply the MBP rules to disjunctions introduced by the QE rules and not to those originally occurring in φ. Correspondingly, we can convert our QE algorithm ARRAYQE to ARRAYQE$_M$ by replacing each rule R with R_M. We can then obtain an MBP ARRAYMBP$(\varphi)(M) = $ ARRAYQE$_M(\varphi)$ and we can show

the following:

Theorem *3:* For any quantifier-free formula φ in ARR, ARRAYMBP$(\exists a : \text{arr}(I, V). \, \varphi)$ is a finite MBP.

The fact that it is an MBP can be easily shown by induction on the number of rewrites applied. The fact that it is finite derives from the fact that there are only finitely many ways to resolve the disjunctions in the QE result.

Moreover, assuming that the evaluation of a formula in a model can be done in polynomial time, we can evaluate ARRAYMBP$(\varphi)(M)$ in time that is polynomial in the size of M and the size of φ. This is because we can polynomially bound the number of times each rule R_M applies, and each rule can only expand the formula size by a constant amount. Fig. 6 shows an example of applying ARRAYMBP.

C. MBP for ARR+LIA

We now consider the combination of the ARR and LIA theories. Assume that the only basic sorts are `bool` and `int`. Furthermore, we only consider linear functions over `int` along with a divisibility predicate (with constant divisors). We developed a finite MBP for LIA in a previous work [17] (call it LIAMBP). When the index sort I is `int`, one can obtain a more efficient MBP with a slight modification of ACKERMANN$_M$ (for eliminating array read terms) that utilizes the predicate symbol $<$. Given a model M of the formula,

$$\exists a \cdot (b = wr(a, i_1, v_1) \vee (rd(wr(a, i_2, v_2), i_3) > 5 \wedge rd(a, i_4) > 0))$$

$$\Leftarrow \exists a \cdot (\boldsymbol{i_2 \neq i_3} \wedge (b = wr(a, i_1, v_1) \vee (\boldsymbol{rd(a, i_3) > 5} \wedge rd(a, i_4) > 0))) \qquad \{\text{WrRd}_M, M \models i_2 \neq i_3\}$$

$$\Leftarrow \exists a \cdot (i_2 \neq i_3 \wedge ((\boldsymbol{a =_{i_1} b} \wedge \boldsymbol{rd(b, i_1) = v_1}) \vee (rd(a, i_3) > 5 \wedge rd(a, i_4) > 0))) \qquad \{\text{PartialEq}; \text{WrEq}_M\}$$

$$\Leftarrow \exists a \cdot \neg(\boldsymbol{a =_{i_1} b}) \wedge i_2 \neq i_3 \wedge (rd(a, i_3) > 5 \wedge rd(a, i_4) > 0) \qquad \{\text{CaseEq}_M, M \not\models a =_{i_1} b\}$$

$$\Leftarrow \exists a, s_3, s_4 \cdot \left(\underbrace{\neg(a =_{i_1} b) \wedge \boldsymbol{i_2 \neq i_3} \wedge (\boldsymbol{s_3 > 5} \wedge \boldsymbol{s_4 > 0})}_{\varphi_2} \right) \\ \wedge \, \boldsymbol{s_3 = rd(a, i_3)} \wedge \boldsymbol{s_4 = rd(a, i_4)} \qquad \{\text{FactorRd}\}$$

$$\Leftarrow \exists a, s_3, s_4 \cdot (\varphi_2 \wedge \neg(\boldsymbol{a =_{i_1} b}) \wedge \boldsymbol{s_3 = rd(a, i_3)} \wedge \boldsymbol{s_4 = rd(a, i_4)}) \qquad \{\text{LiftEqDiseqRd}\}$$

$$\Leftarrow \exists a, s_3, s_4 \cdot (\varphi_2 \wedge s_3 = rd(a, i_3) \wedge s_4 = rd(a, i_4)) \qquad \{\text{ElimDiseq}\}$$

$$\Leftarrow \exists s_3, s_4 \cdot (\varphi_2 \wedge (\boldsymbol{i_3 = i_4} \wedge \boldsymbol{s_3 = s_4})) \qquad \{\text{Ack}_M, M \models i_3 = i_4\}$$

Fig. 6: Illustrating ArrayMBP on the example of Fig. 4 with a given model M.

$$\text{MbpLeft} \quad \frac{\varphi[\psi_1 \vee \psi_2] \qquad M \models \varphi, \psi_1}{\varphi[\psi_1]}$$

$$\text{MbpRight} \quad \frac{\varphi[\psi_1 \vee \psi_2] \qquad M \models \varphi, \psi_2}{\varphi[\psi_2]}$$

$$\text{MbpVac} \quad \frac{\varphi[\psi_1 \vee \psi_2] \qquad M \models \varphi \qquad M \not\models \psi_1, \psi_2}{\varphi[\bot]}$$

Fig. 5: MBP rules for formulas in negation-normal form.

one can first partition the set of index terms t_k's according to their interpretations in M and choose a representative for each equivalence class. Then, the conjunction in the result of the rule is modified as follows: (a) for every equivalence class, add the equality $t_k = t_\ell$ for every non-representative t_ℓ, where t_k is the representative, (b) linearly order the representatives and add the corresponding inequalities. The modified rule (and hence, the resulting MBP) is linear in time and space.

However, the combination of arrays and integers introduces terms over the combined signature which need to be handled as well. For example, there is no equivalent quantifier-free formula for $\exists i : \text{int} \cdot rd(a, i) > 0$. This implies that there does not exist a finite MBP for the combination of LIA and ARR. In the example, the only way to under-approximate the quantification is to use the substitution method, replacing i with its interpretation in a model $M \models rd(a, i) > 0$ as a numeric literal.

Based on the above observations, we obtain an MBP for ARR+LIA as follows. First, we apply ArrayMBP, using the modified Ackermann$_M$ above, to eliminate array quantifiers. Then, we use LiaMBP to eliminate integer quantifiers that do not appear in any array term. Finally, we use the substitution method to eliminate any remaining integer quantifiers. When the last step of substitution method is not necessary, the resulting MBP will be finite.

IV. The Compositional Verification Framework

MBP plays a crucial role in enabling the search for compositional proofs. In this section, we will consider the role played by MBP in a model checking framework called Spacer [17]. In this framework, MBP is used to create succinct localized proof sub-goals that make it possible to reason about only one procedure at a time. The proof goals take the form of under-approximate summaries, either of the calling context of a procedure or of the procedure itself. Without some form of projection, Spacer would not be compositional, as it would build up formulas of exponential size, in effect inlining procedures to create bounded model checking formulas.

A. Modeling programs with CHCs

Spacer checks safety of procedural programs by reducing the problem to SMT of a special kind of formulas known as *Constrained Horn Clauses* (CHCs) [5], [17], [14]. We augment the signature $\mathcal{S}$ with a set of fresh predicate symbols $\mathcal{P}$. A *Constrained Horn Clause* (CHC) is a formula of the form

$$\forall \overline{x} \cdot \underbrace{\bigwedge_{k=1}^{m} P_k(\overline{x}_k) \wedge \varphi(\overline{x})}_{body} \implies head$$

where for each k, P_k is a symbol in $\mathcal{P}$, $\overline{x}_k \subseteq \overline{x}$ and $|\overline{x}_k|$ is equal to the arity of P_k. The constraint φ is a formula over $\mathcal{S}$, and $head$ is either an application of a predicate in $\mathcal{P}$ or another formula over $\mathcal{S}$. We use $body$ to refer to the antecedent of the CHC, as shown above. A CHC is called a *query* if $head$ is a formula over $\mathcal{S}$ and otherwise, it is called a *rule*. If $m \leq 1$ in the body, the CHC is *linear* and is *non-linear* otherwise. Following the convention of logic programming literature, we also write the above CHC as $head \leftarrow P_1(\overline{x}_1), \ldots, P_m(\overline{x}_m), \varphi(\overline{x})$.

Intuitively, each predicate symbol P_k represents an unknown partial correctness specification of a procedure (that is, an over-approximate summary). A query defines a property to be proved, while each rule gives modular verification condition for one procedure. A satisfying assignment to the symbols P_k

is thus a certificate that the program satisfies its specification and corresponds to the annotations in a Floyd/Hoare style proof. In this work, we are interested in finding annotations that can be expressed in the *quantifier-free* fragment of our first-order language, to avoid the difficulty of reasoning with quantifiers.

Any given set of CHCs encoding safety of procedural programs can be transformed to an equisatisfiable set of just three CHCs with a single predicate symbol (encoding the program location using a variable). These CHCs have the following form:

$$Inv(\overline{x}) \leftarrow init(\overline{x}) \qquad \neg bad(\overline{x}) \leftarrow Inv(\overline{x})$$
$$Inv(\overline{x}') \leftarrow Inv(\overline{x}), Inv(\overline{x}^o), tr(\overline{x}, \overline{x}^o, \overline{x}') \tag{4}$$

Intuitively, Inv is the program invariant, $\overline{x}$ denotes the pre-state of a program transition, $\overline{x}'$ denotes the post-state, and $\overline{x}^o$ denotes the summary of a procedure call (if one is made). If there are no procedure calls, tr is independent of $\overline{x}^o$ and $Inv(\overline{x}^o)$ can be dropped: in this case Inv denotes an inductive invariant of an ordinary transition system. In the sequel, we restrict to this normal form and consider only quantifier-free interpretations of the predicate Inv.

It is useful to rewrite the above rules using a function $\mathcal{F}$ that substitutes given predicates $\phi_A(\overline{x})$ and $\phi_B(\overline{x})$ for the occurrences of Inv in the rule bodies. That is, let

$$\mathcal{F}(\varphi_A, \varphi_B) \equiv (\varphi_A(\overline{x}) \wedge \varphi_B(\overline{x}^o) \wedge tr(\overline{x}, \overline{x}^o, \overline{x}'))$$
$$\vee init(\overline{x}')$$

The rules are thus equivalent to $\mathcal{F}(Inv, Inv) \Rightarrow Inv(\overline{x})$. Abusing notation, we will also write $\mathcal{F}(\varphi_A)$ for $\mathcal{F}(\varphi_A, \varphi_A)$.

B. The SPACER framework

SPACER is a general framework that can be instantiated for a given logical theory T by supplying three elements: *(a)* a model-generating SMT solver for T, *(b)* an MBP procedure MBP for T and *(c)* in interpolation procedure ITP for T. Compared to other SMT-based algorithms (e.g., [3], [13], [10], [18]), the key distinguishing feature of SPACER is compositional reasoning. That is, instead of checking satisfiability of large formulas generated by program unwinding, SPACER iteratively creates and checks local reachability queries for individual procedures. In this way it is similar to IC3 [6], [9], a SAT-based algorithm for safety of finite-state transition systems, and GPDR [16], its extension to Linear Real Arithmetic. Like these methods, SPACER maintains a sequence of over-approximations of procedure behaviors, called *may summaries*, corresponding to program unwindings. However, unlike other approaches, SPACER also maintains under-approximations of procedure behaviors, called *must summaries*, to avoid redundant reachability queries. Another distinguishing feature of SPACER is the use of MBP for efficiently handling existentially quantified formulas to create a new query or a must summary. We note, however, that MBP is a general technique and can be exploited in IC3/PDR as well.[2]

---

[2] Arguably sub-goal creation in IC3 is a simple MBP for propositional logic.

Alg. 2 gives a simplified description of SPACER as a solver for CHCs in the form of (4) (though SPACER handles general CHCs). It is described using a set of rules that can be applied non-deterministically. Each rule is presented as a guarded command "[*grd*] *cmd*", where *cmd* can be executed only if *grd* holds.

Input: Formulas $init(\overline{x}), tr(\overline{x}, \overline{x}^o, \overline{x}'), bad(\overline{x})$
Output: Inductive invariant (FO interpretation of Inv satisfying (4)) or UNSAFE

if $(init \wedge bad)$ *satisfiable* **then return** UNSAFE
// initialize data structures
$Q := \emptyset$ *// set of pairs $\langle\varphi, i\rangle, i \in \mathbb{N}$*
$N := 0$ *// max level, or recursion depth*
$\mathcal{O}_0 = init, \mathcal{O}_i = \top, \forall i > 0$ *// may summary sequence*
$\mathcal{U} = init$ *// must summary*
forever *non-deterministically* **do**
  (Candidate) [$(\mathcal{O}_N \wedge bad)$ *satisfiable*]
    $Q := Q \cup \langle\varphi, N\rangle$, for some $\varphi \implies \mathcal{O}_N \wedge bad$
  (DecideMust) [$\langle\varphi, i+1\rangle \in Q, M \models \mathcal{F}(\mathcal{O}_i, \mathcal{U}) \wedge \varphi'$]
    $Q := Q \cup \langle\text{MBP}(\exists \overline{x}^o, \overline{x}' \cdot \mathcal{F}(\mathcal{O}_i, \mathcal{U}) \wedge \varphi', M), i\rangle$
  (DecideMay) [$(\varphi, i+1) \in Q, M \models \mathcal{F}(\mathcal{O}_i) \wedge \varphi'$]
    $Q := Q \cup \langle\text{MBP}(\exists \overline{x}, \overline{x}' \cdot \mathcal{F}(\mathcal{O}_i) \wedge \varphi', M)[\overline{x}/\overline{x}^o], i\rangle$
  (Leaf) [$(\varphi, i) \in Q, \mathcal{F}(\mathcal{O}_{i-1}) \implies \neg\varphi', i < N$]
    $Q := Q \cup \langle\varphi, i+1\rangle$
  (Successor) [$\langle\varphi, i+1\rangle \in Q, M \models \mathcal{F}(\mathcal{U}) \wedge \varphi'$]
    $\mathcal{U} := \mathcal{U} \vee \text{MBP}(\exists \overline{x}, \overline{x}^o \cdot \mathcal{F}(\mathcal{U}) \wedge \varphi', M)[\overline{x}/\overline{x}']$
  (Conflict) [$\langle\varphi, i+1\rangle \in Q, \mathcal{F}(\mathcal{O}_i) \implies \neg\varphi'$]
    $\mathcal{O}_j := \mathcal{O}_j \wedge \text{ITP}(\mathcal{F}(\mathcal{O}_i), \neg\varphi')[\overline{x}/\overline{x}'], \forall j \leq i+1$
  (Induction) [$(\varphi \vee \psi) \in \mathcal{O}_i, \mathcal{F}(\varphi \wedge \mathcal{O}_i) \implies \varphi'$]
    $\mathcal{O}_j := \mathcal{O}_j \wedge \varphi, \forall j \leq i+1$
  (Unfold) [$\mathcal{O}_N \implies \neg bad$] $N := N+1$
  (Safe) [$\mathcal{O}_{i+1} \implies \mathcal{O}_i$] **return** invariant $\mathcal{O}_i$
  (Unsafe) [$(\mathcal{U} \wedge bad)$ *satisfiable*] **return** UNSAFE

Algorithm 2: Rule-based description of SPACER.

As shown in Alg. 2, SPACER maintains a set of reachability queries Q, a sequence of may summaries $\{\mathcal{O}_i\}_{i\in\mathbb{N}}$, and a must summary $\mathcal{U}$. Intuitively, a query $\langle\varphi, i\rangle$ corresponds to checking if φ is reachable for recursion depth i, $\mathcal{O}_i$ over-approximates the reachable states for recursion depth i, and $\mathcal{U}$ under-approximates the reachable states. N denotes the current bound on recursion depth. The sequence of may summaries and N correspond to the *trace of approximations* and the maximum *level* in IC3/PDR, respectively. For convenience, let $\mathcal{O}_{-1}$ be $\bot$. $\text{MBP}(\varphi, M)$, for a formula $\varphi = \exists \overline{v} \cdot \varphi_{qf}$ and model $M \models \varphi_{qf}$, denotes the result of some MBP function associated with φ for the model M.

Alg. 2 initializes N to 0 and, $\mathcal{O}_0$ and $\mathcal{U}$ to $init$. **Candidate** initiates a backward search for a counterexample beginning with a set of states in bad. The potential counterexample is expanded using either **DecideMust** or **DecideMay**. **DecideMust** *jumps over* the call $Inv(\overline{x}^o)$, in the last CHC of (4), utilizing the must summary $\mathcal{U}$. **DecideMay**, on the other hand, creates a query for the call using the may summary of its calling context.

Successor updates $\mathcal{U}$ when a query is known to be reachable. The other rules are similar to IC3 [6] and GPDR [16] and we skip their explanation in the interest of space. SPACER is sound and if MBP utilizes finite MBP functions, SPACER also terminates for a fixed N [17].

C. Instantiation for ARR+LIA

In instantiating this framework for ARR+LIA, the key ingredient is the MBP procedure of the previous section. An interpolation procedure ITP can be trivially obtained by using literal-dropping approach based on UNSAT cores, or a more sophisticated approach can be taken (e.g., see [16], [18]).

Because we do not have a *finite* MBP, SPACER is not guaranteed to terminate even for a fixed bound on the recursion depth N. That is, it can generate an infinite sequence of queries and must summaries. Note that MBP is used in 3 rules: **DecideMay**, **DecideMust**, and **Successor**. The elimination of quantifiers in **Successor** is only an optimization and can be avoided. This is not the case with **DecideMay** or **DecideMust** without changing the structure of the queries, the considerations of which are outside the scope of this paper. In the following, we identify restrictions on the CHCs where termination is still guaranteed and for the other cases, we propose some heuristic modifications to MBP and ITP to help avoid divergence.

1) Equality resolution in MBP*:* There are several cases where terms over combined signatures appear in conjunction with equality terms over the index quantifier, e.g., $\exists i : \text{int} \cdot i = t \wedge rd(a, i) > 0$ for a term t independent of i. In these cases, the quantifier can be eliminated using equality resolution, e.g., $rd(a, t) > 0$ in the above example. Such cases seem to be natural in the case of a single procedure, i.e., when tr in (4) is independent of $\overline{x}^o$. Consider a disjunct δ in a DNF representation of tr. Now, δ represents a path in the procedure and typically, index terms (in reads and writes) in δ can be ordered such that every index term is a function of the previous index terms or the current-state variables $\overline{x}$. This makes it possible to eliminate any index variables in $\overline{x}'$ using equality resolution as mentioned above.

2) Privileging array equalities: Here is a simple example that exhibits non-termination:

$$Inv(a, b) \leftarrow a = b$$
$$\bot \leftarrow Inv(a, b), rd(a, j) < 0, rd(b, j) > 0$$

Here, intuitively, $Inv(a, b)$ denotes the summary of a procedure which takes an array a as input and produces b as output and we are interested in checking if there is sign change in the value at an index j as a result of the procedure call. For this example, **DecideMay** creates queries of the form $rd(a, k) < 0 \wedge rd(b, k) > 0$ where k is a specific integer constant. If ITP returns interpolants of the form $rd(a, k) = rd(b, k)$, it is easy to see that SPACER would not terminate even for $N = 0$, even though there is a trivial solution: $a = b$.

To alleviate this problem, we modify MBP and ITP to promote the use of array equalities in interpolants. Let ψ be the result of MBP for a given model M. For every pair of array terms a, b in ψ, we strengthen ψ with the array equality $a = b$ or disequality $a \neq b$, depending on whether $M \models a = b$ holds or not. In the above example, the queries will now be of the form $rd(a, k) < 0 \wedge rd(b, k) > 0 \wedge a \neq b$. However, $rd(a, k) = rd(b, k)$ continues to be an interpolant whereas the desired interpolant is $a = b$. To reduce the dependence on specific integer constants in the learned interpolants, and hence in the may summaries, we modify ITP as follows. Suppose we are computing an interpolant for $\psi \implies \neg\varphi'$ (as occurs in **Conflict**). We let $\varphi = \varphi_1 \wedge \varphi_2$ where φ_2 contains all the literals where an integer quantifier is substituted using its interpretation in a model. Using a *minimal unsatisfiable subset* (MUS) algorithm, we can generalize φ_2 to $\hat{\varphi}_2$ such that $\psi \wedge (\varphi_1 \wedge \hat{\varphi}_2)'$ is unsatisfiable and then obtain ITP$(\psi, \neg (\varphi_1 \wedge \hat{\varphi}_2)')$. In the above example, for $N = 0$ we have $\psi = (a = b)$, $\varphi_1 = (a \neq b)$, and $\varphi_2 = rd(a, k) < 0 \wedge rd(b, k) > 0$. One can show that $\hat{\varphi}_2$ is simply $\top$ and the only possible interpolant is $a = b$. In our implementation, we add such (dis-)equalities on-demand in a lazy fashion. Note that adding such (dis-)equalities to the queries is only a heuristic and may not always help with termination.

V. EXPERIMENTAL RESULTS

As noted in the introduction, the array theory allows us to model heap references accurately. This eliminates the need to inline procedures so that heap-allocated objects are reduced to local variables. We hypothesize that the resulting increase in modularity will allow SPACER to more efficiently verify procedural programs using ARRAYMBP, in spite of the potential for divergence due to non-finiteness of the MBP.

We test this hypothesis using a prototype implementation of SPACER with ARRAYMBP.[3] To verify C programs, we use SEAHORN [14], which uses the LLVM infrastructure to compile and optimize the input program, then encodes the verification conditions as CHCs in the SMT-LIB2 format. SEAHORN can optionally inline procedure calls before encoding, allowing us to test our hypothesis regarding modularity.

For reference, we also compare SPACER to the implementation of GPDR [16] in Z3 [8]. A key difference between SPACER and GPDR is that the latter does not use must summaries. Z3 also uses MBP, but is limited to equality resolution and the substitution method. As a result Z3 GPDR is effective only for inlined programs.

We use benchmarks from the software verification competition SVCOMP'15 [4]. We considered the 215 benchmarks from the *Device Drivers* category where Z3 GPDR (with inlining) needed more than a minute of runtime or did not terminate within the resource limits of SVCOMP [15]. All experiments have been carried out using a 2.2 GHz AMD Opteron(TM) Processor 6174 and 516GB RAM, running Ubuntu Linux. Our resource limits are 30 minutes and 15GB for each verification task. In the scatter plots that follow, a diamond indicates a *time-out*, a star indicates a *mem-out*, and a box indicates an anomaly in the implementation.

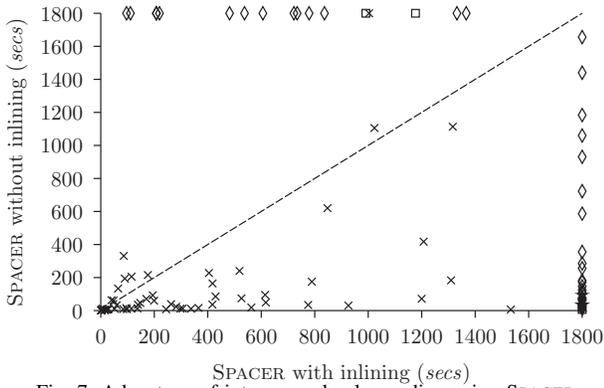
[3]https://bitbucket.org/spacer/code

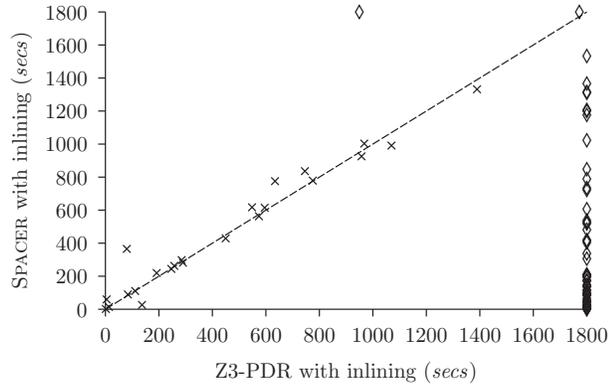Fig. 7: Advantage of inter-procedural encoding using SPACER.

Fig. 8: SPACER vs. Z3 on hard SVCOMP benchmarks with inlining.

The scatter plot in Fig. 7 compares the combined run time for the CHC encoding and verification, when inlining is turned on and off. A clear advantage is seen in the non-inlining case. This shows that SPACER is able to effectively exploit the additional modularity that is made possible by ARRAYMBP, and that this advantage outweighs any occurrences of divergence due to non-finite MBP.[4] We note that SPACER with only LIA is able to handle only a small fraction of the non-inlined benchmarks. This result confirms our hypothesis.

For reference, we also compare to the performance of Z3 GPDR. We observed that without ARRAYMBP, Z3 is very ineffective in the non-inlined case. We should mention, however, that of the 7 unsafe programs verified by Z3, 5 could not be verified by SPACER. Fig. 8 compares SPACER and Z3 with inlining on. This shows an overwhelming advantage for SPACER, which is due to its more effective MBP approach.

VI. RELATED WORK

There are several SMT-based approaches for sequential program verification that iteratively check satisfiability of formulas corresponding to safety of various unwindings of the program [3], [13], [10], [18]. However, these monolithic SMT formulas can grow exponentially. In contrast, the SPACER framework [17] we use allows us to do a compositional proof search for safety. Such local proof search is also found in the IC3 algorithm for hardware model checking [6] and its extensions to software model checking (e.g., [16]), although

---

[4]Unfortunately, we have no way to distinguish divergence from timeouts.

SPACER is the first to use under-approximate summaries of procedures for avoiding redundant proof sub-goals. Model-based generalizations have also been used to obtain projections efficiently in decision procedures for quantified formulas [19].

VII. CONCLUSION AND FUTURE WORK

We have presented a procedure for existentially projecting array variables from formulas over combined theories of ARR, LIA, and propositional logic. We have adapted the procedure to a finite MBP for array variables. While existential projection is worst-case exponential, the corresponding MBP is polynomial. However, projecting arrays might introduce new existentially quantified variables (whose sort is the same as the index- or value-sort of the eliminated array). For projecting these variables, a finite MBP need not exist. We described heuristics for obtaining a practical (but not necessarily finite) MBP procedure, obtaining an instantiation of the SPACER framework for verification of safety of sequential heap-manipulating programs. We show that the new variant of SPACER is effective for constructing compositional proofs of Linux Device Drivers. In the future, we plan to extend these ideas for handling more complex heap-manipulating programs that require universal quantifiers in the program invariants.

REFERENCES

[1] W. Ackermann, *Solvable Cases of The Decision Problem*. North-Holland, Amsterdam, 1954.
[2] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "From Under-Approximations to Over-Approximations and Back," in *TACAS*, 2012.
[3] ——, "Whale: An Interpolation-Based Algorithm for Inter-procedural Verification," in *VMCAI*, 2012.
[4] D. Beyer, "Software Verification and Verifiable Witnesses – (Report on SV-COMP 2015)," in *TACAS*, 2015.
[5] N. Bjørner, K. McMillan, and A. Rybalchenko, "Program Verification as Satisfiability Modulo Theories," in *SMT*, 2012.
[6] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *VMCAI*, 2011.
[7] W. Craig, "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory," *Symbolic Logic*, vol. 22(3), 1957.
[8] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, 2008.
[9] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient Implementation of Property Directed Reachability," in *FMCAD*, 2011.
[10] M. H. et al., "Ultimate Automizer with SMTInterpol - (Competition Contribution)," in *TACAS*, 2013.
[11] G. Fedyukovich, O. Sery, and N. Sharygina, "eVolCheck: Incremental Upgrade Checker for C," in *TACAS*, 2013.
[12] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based Unbounded Symbolic Model Checking Using Circuit Cofactoring," in *ICCAD*, 2004.
[13] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing Software Verifiers from Proof Rules," in *PLDI*, 2012.
[14] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. Navas, "The SeaHorn Verification Framework," in *CAV*, 2015.
[15] A. Gurfinkel, T. Kahsai, and J. A. Navas, "SeaHorn: A Framework For Verifying C Programs - (Competition Contribution)," in *TACAS*, 2015.
[16] K. Hoder and N. Bjørner, "Generalized Property Directed Reachability," in *SAT*, 2012.
[17] A. Komuravelli, A. Gurfinkel, and S. Chaki, "SMT-Based Model Checking for Recursive Programs," in *CAV*, 2014.
[18] K. L. McMillan and A. Rybalchenko, "Solving Constrained Horn Clauses using Interpolation," Tech. Rep. MSR-TR-2013-6, 2013.
[19] D. Monniaux, "Quantifier Elimination by Lazy Model Enumeration," in *CAV*, 2010.
[20] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt, "A Decision Procedure for an Extensional Theory of Arrays," in *LICS*, 2001.