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 \( S \) contains basic sorts (e.g., \( \text{bool} \) of Booleans, \( \text{int} \) of integers, etc.) and array sorts. An array sort \( \text{arr}(I,V) \) 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 $\text{rd} : \text{arr}(I, V) \times I \rightarrow V$ and $\text{wr} : \text{arr}(I, V) \times I \times V \rightarrow \text{arr}(I, V)$ for reading from and writing to the array. Intuitively, $\text{rd}(a, i)$ denotes the value stored in the array $a$ at the index $i$ and $\text{wr}(a, i, v)$ 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**

$$\forall a : \text{arr}(I, V) \forall i, j : I \forall v : V$$

$$(i = j \implies \text{rd}(\text{wr}(a, i, v), j) = v) \land$$

$$(i \neq j \implies \text{rd}(\text{wr}(a, i, v), j) = \text{rd}(a, j))$$

**Extensionality**

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

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 $\text{rd}(a, j)$ 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 $\overline{\pi} : S$ to denote that every term in vector $\pi$ has sort $S$, $\overline{\pi}(k)$ to denote the $k$th component of $\pi$, and $y \in \overline{\pi}$ to denote that $y$ is equal to some component of $\overline{\pi}$, i.e., $\bigvee_{k=1}^{\overline{\pi}} y = \overline{\pi}(k)$. Let $\overline{i} : I$ and $\overline{v} : V$ be vectors of index and value terms of the same length $m$. We write $\text{wr}(\overline{a}, \overline{i}, \overline{v})$ to denote $\text{wr}(\text{wr}(\ldots, \text{wr}(a, \overline{i}(0), \overline{v}(0)), \ldots), \overline{i}(m), \overline{v}(m))$. Unless specified otherwise, $S$ 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 $\forall j : I \cdot (j \notin \overline{i} \implies \text{rd}(a, j) = \text{rd}(b, j))$ and call such formulas **partial equalities** [20]. Using extensionality, one can easily show the following

$$a =_{\varnothing} b \equiv a = b$$ (1)

$$\text{wr}(a, j, v) =_{\overline{i}} b \equiv (j \in \overline{i} \land a =_{\overline{i}} b) \lor$$ (2)

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

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

Given formulas $\varphi_A(\overline{\tau}, \overline{\pi})$ and $\varphi_B(\overline{\gamma}, \overline{\pi})$ with $\overline{\tau} \cap \overline{\gamma} = \emptyset$ and $\varphi_A \implies \varphi_B$, a Craig Interpolant [7], denoted $\text{1TP}(\varphi_A, \varphi_B)$, is a formula $\varphi_I(\overline{\pi})$ such that $\varphi_A \models \varphi_I$ 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 $\varphi$ of the form $\exists \overline{\pi} \cdot \varphi_{qf}(\overline{\tau}, \overline{\gamma})$ where $\varphi_{qf}$ is quantifier-free. The problem of quantifier elimination (QE) in $\varphi$ is to find a logically equivalent quantifier-free formula $\psi(\overline{\gamma})$. In this case, we say that $\psi$ is the result of projecting $\overline{\pi}$ in $\varphi_{qf}$.

A model-based projection (MBP) for $\varphi$ is an operator $\text{Proj}$ that takes a model $M$ of $\varphi_{qf}$ and returns a quantifier-free formula $\psi_M(\overline{\gamma})$ such that $M \models \psi_M$ and $\psi_M$ entails $\varphi$. The operator $\text{Proj}$ 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 $\text{Proj}$. We will refer to $\text{Proj}(M)$ 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 $\varphi$ 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 $\varphi$ is quantifier-free. 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 $I$ or $V$.

As a simplification, we restrict the interpretations of $I$, the index sort, to infinite domains. Handling finite index domains requires a slight adaptation of the algorithms as described in Appendix A.

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

```plaintext
1 $\varphi_1 \leftarrow (\text{ElimWf}^\ast)(\exists a \cdot \varphi)$
2 $\varphi_2 \leftarrow (\text{CaseSplitEq}^\ast; \text{FactorRd}^\ast)(\varphi_1)$
3 $\text{(\bigvee_{k=1}^{n} \delta_{k})} \leftarrow \text{LiftEqDiseqRd}(\varphi_2)$
4 for $k \in [1, n]$ do
5 $\psi_{k} \leftarrow (\text{ElimEq}^\ast; \text{ElimDiseq}^\ast; \text{Ackermann})\text{(}\delta_{k})$
6 return $\bigvee_{k=1}^{n} \psi_{k}$
```

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

\(^1\) MBP as defined in [17] corresponds to finite MBP here.
rewrites the formula above the line to the logically equivalent formula below the line. We use regular expression notation to express sequences of rewrites. In particular, Kleene star does not appear in a and \( \overline{a} \) denotes fresh variables

\[
\exists a \cdot \varphi[a = \overline{t} t \wedge \varphi[\overline{T}] \lor (\neg(a = \overline{t} t) \wedge \varphi[\overline{1}])]
\]

ELIMEQ
\[
\exists a \cdot (a = \overline{t} t \wedge \varphi) \leadsto \exists a \cdot \varphi[\text{wr}(t, \overline{1}, \overline{1})/a]
\]

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

\[
\exists a \cdot (\varphi \wedge \bigwedge_{k=1}^{m} \neg(a = \overline{t} t_k))
\]

ELIMDISEQ

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

\[
\exists a \cdot (\varphi \wedge \bigwedge_{k=1}^{m} \neg(a = \overline{t} t_k))
\]

ACKERMANN
\[
\varphi \wedge \bigwedge_{1 \leq k < t \leq m} (t_k = t_e \implies s_k = s_e)
\]

where \( m \in \mathbb{N} \) and \( a \) does not appear in \( \varphi \), \( s_k \)’s, or \( t_k \)’s

Fig. 3: Rewriting rules for QE of arrays.
\[ \exists a \cdot (b = \varphi(a, i_1, v_1) \vee (rd(a, i_2, v_2) > 5 \land rd(a, i_4) > 0)) \]

\[ \equiv \exists a \cdot (i_2 = i_3 \land (b = \varphi(a, i_1, v_1) \vee (v_2 > 5 \land rd(a, i_4) > 0)) \vee (i_2 \neq i_3 \land (b = \varphi(a, i_1, v_1) \vee (rd(a, i_3) > 5 \land rd(a, i_4) > 0))) \]

\[ \equiv \exists a \cdot (i_2 = i_3 \land ((a = i_1 \land \neg rd(b, i_1) = v_1) \vee (v_2 > 5 \land rd(a, i_4) > 0)) \vee (a = i_1 \land \neg rd(b, i_1) = v_1 \vee (v_2 > 5 \land rd(a, i_4) > 0))) \]

\[ \equiv \exists a \cdot (\neg(a = i_1) \land \neg (i_2 = i_3 \land rd(b, i_1) = v_1) \vee (i_2 \neq i_3 \land (rd(a, i_3) > 5 \land rd(a, i_4) > 0))) \]

\[ \equiv \exists a, s_3, s_4 \cdot (\neg(a = i_1) \land \neg (i_2 = i_3 \land rd(b, i_1) = v_1) \vee (i_2 \neq i_3 \land (rd(a, i_3) > 5 \land rd(a, i_4) > 0))) \]

\[ \land s_3 = rd(a, i_3) \land s_4 = rd(a, i_4) \]

\[ \equiv \exists a, s_3, s_4 \cdot (\varphi_1 \land a = i_1) \land \neg s_3 = rd(a, i_3) \land \neg s_4 = rd(a, i_4) \]

\[ \equiv \exists a, s_3, s_4 \cdot (\varphi_2 \land \neg(a = i_1) \land \neg s_3 = rd(a, i_3) \land \neg s_4 = rd(a, i_4)) \]

\[ \equiv \exists a, s_3, s_4 \cdot (\varphi_1 \land s_3 = rd(a, i_3) \land s_4 = rd(a, i_4)) \land \varphi_2 \land \neg(a = i_1) \land \neg s_3 = rd(a, i_3) \land \neg s_4 = rd(a, i_4) \]

\[ \equiv \exists a, s_3, s_4 \cdot (\varphi_1 \land s_3 = rd(a, i_3) \land s_4 = rd(a, i_4)) \land \varphi_2 \land \neg(a = i_1) \land (s_3 = i_4 \implies s_3 = s_4) \]

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 \( \varphi \). If we also have \( M \models \psi_1 \), then \( M \models \varphi \psi_2 \) and clearly, \( \varphi \psi_1 \) entails \( \varphi \). This gives us a way to eliminate a disjunction while preserving a given model and maintaining an under-approximation. If neither \( \psi_1 \) nor \( \psi_2 \) is true in \( M \), we can similarly replace \( \varphi \) with \( \varphi \psi_1 \psi_2 \). These transformations are expressed as MBP rewrite rules in Fig. 5.

For each QE rule \( R \), we can produce a corresponding under-approximation rule \( R_M \) that preserves model \( M \). This rule can be written \( R : (\text{MBPLEFT} \mid \text{MBPRIGHT} \mid \text{MBPVAC})^* \). 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 \( \varphi \). 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 \( \text{ARRAYMBP}(\varphi)(M) = \text{ARRAYQE}_M(\varphi) \) and we can show the following:

**Theorem 3**: For any quantifier-free formula \( \varphi \) in ARR, \( \text{ARRAYMBP}(\exists a : \text{ARR}(I, \varphi). \varphi) \) 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 \( \text{ARRAYMBP}(\varphi)(M) \) in time that is polynomial in the size of \( M \) and the size of \( \varphi \). 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 \text{bool} and \text{int}. Furthermore, we only consider linear functions over \text{int} along with a divisibility predicate (with constant divisors). We developed a finite MBP for LIA in a previous work [17] (call it LIAMB). When the index sort \( I \) is \text{int}, one can obtain a more efficient MBP with a slight modification of \( \text{ACKERMANN}_M \) (for eliminating array read terms) that utilizes the predicate symbol \( < \). Given a model \( M \) of the formula,
∃a : (b = wr(a, i_1, v_1) ∨ (rd(wr(a, i_2, v_2), i_3) > 5 ∧ rd(a, i_4) > 0))

⇐ \exists a : (i_2 ≠ i_3 ∧ (b = wr(a, i_1, v_1) ∨ (rd(a, i_3) > 5 ∧ rd(a, i_4) > 0))) [WRDA, M \models i_2 ≠ i_3]

⇐ \exists a : (i_2 ≠ i_3 ∧ ((a = i_1 ∧ b ∨ rd(b, i_1) = v_1) ∨ (rd(a, i_3) > 5 ∧ rd(a, i_4) > 0))) [PARTIALEQWRDA, M \models i_2 ≠ i_3]

⇐ \exists a : ¬(a = i_1) \land \exists a : (i_2 ≠ i_3 ∧ (rd(a, i_3) > 5 ∧ rd(a, i_4) > 0)) [CASEEQM, M \models a = i_1, b]

⇐ \exists a, s_3, s_4 : (¬(a = i_1) \land i_2 ≠ i_3 ∧ (s_3 > 5 ∧ s_4 > 0)) [FACTORRD]

⇐ \exists a, s_3, s_4 : (\varphi_2 \land ¬(a = i_1) \land s_3 = rd(a, i_3) \land s_4 = rd(a, i_4)) [LIFTEQDISEQRD]

⇐ \exists a, s_3, s_4 : (\varphi_2 ∧ ¬(a = i_1) ∧ s_3 = rd(a, i_3) ∧ s_4 = rd(a, i_4)) [ELIMDISEQ]

⇐ \exists s_3, s_4 : (\varphi_2 ∧ (i_3 = i_4 ∧ s_3 = s_4)) [ACKM, M \models i_3 = i_4]

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

**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 S with a set of fresh predicate symbols P. A Constrained Horn Clause (CHC) is a formula of the form

\[ \forall \mathbf{x} : \bigwedge_{k=1}^{m} P_k(\mathbf{x}_k) \land \varphi(\mathbf{x}) \implies \text{head} \]

where for each k, P_k is a symbol in P, \( \mathbf{x}_k \subseteq \mathbf{x} \) and \( |\mathbf{x}_k| \) is equal to the arity of P_k. The constraint \( \varphi \) is a formula over S, and head is either an application of a predicate in P or another formula over 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 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 ← P_1(\mathbf{x}_1), \ldots, P_m(\mathbf{x}_m), \varphi(\mathbf{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:

\[ \text{Inv}(\tau) \leftarrow \text{init}(\tau) \quad \neg \text{bad}(\tau) \leftarrow \text{Inv}(\tau) \]

\[ \text{Inv}(\tau') \leftarrow \text{Inv}(\tau), \text{Inv}(\tau''), \text{tr}(\tau, \tau', \tau'') \] (4)

Intuitively, Inv is the program invariant, \( \tau \) denotes the pre-state of a program transition, \( \tau' \) denotes the post-state, and \( \tau'' \) denotes the summary of a procedure call (if one is made). If there are no procedure calls, \( \text{tr} \) is independent of \( \tau'\) and \( \text{Inv}(\tau'') \) 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 approximations of procedure behaviors, called \( \text{CHCs} \). These \( \text{CHCs} \) have the following form:

\[ \mathcal{F}(\varphi_A, \varphi_B) \equiv (\varphi_A(\tau) \land \varphi_B(\tau') \land \text{tr}(\tau, \tau', \tau'')) \land \neg \text{init}(\tau') \]

The rules are thus equivalent to \( \mathcal{F}(\text{Inv}, \text{Inv}) \Rightarrow \text{Inv}(\tau) \). 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 model procedure \( \text{MBP} \) for \( T \) and (c) in interpolation procedure \( \text{ITP} \) 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 \textit{may summaries}, corresponding to program unwindings. However, unlike other approaches, Spacer also maintains under-approximations of procedure behaviors, called \textit{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.\(^2\)

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 “[ \text{cmd} ]”, where \text{cmd} can be executed only if \text{cond} holds.

**Input:** Formulas \( \text{init}(\tau), \text{tr}(\tau, \tau', \tau''), \text{bad}(\tau) \)

**Output:** Inductive invariant (FO interpretation of Inv satisfying (4)) or \text{UNSAFE}

If \( (\text{init} \land \text{bad}) \) satisfiable then return \text{UNSAFE}

// initialize data structures
\[ Q := \emptyset \quad \text{N := 0} \quad \text{O := init} \quad \text{U := init} \]

forever non-deterministically do

\[ (\text{Candidate}) \ [ (\text{O} \land \text{bad}) \text{ satisfiable }] \]

\[ Q := Q \cup \langle \varphi, N \rangle, \text{for some } \varphi \iff \text{O} \land \text{bad} \]

\[ (\text{DecideMust}) \ [ (\langle \varphi, i+1 \rangle \in Q, \text{M} \models \text{F(O),U} \land \varphi'] ] \]

\[ \text{U} := \text{U} \lor \text{MBP(\tau', \tau'')} \land \text{F(O),U} \land \varphi', \text{M}[\tau'/\tau''] \]

\[ \text{O} := \text{O} \lor \text{ITP(\tau', \tau'')} \land \neg \varphi', \text{U} \land \varphi', \text{M}[\tau'/\tau''] \]

\[ \text{(Conflict)} \ [ (\langle \varphi, i+1 \rangle \in Q, \text{F(O,i)} \models \neg \varphi'] ] \]

\[ \text{O} := \text{O} \land \varphi, \text{U} \land \varphi, \text{M}[\tau'/\tau''] \]

\[ \text{(Induction)} \ [ (\varphi \land \psi) \in \text{O}, \text{F(\varphi \land \psi)} \models \varphi'] ] \]

\[ \text{O} := \text{O} \land \varphi, \text{U} \land \varphi, \text{M}[\tau'/\tau''] \]

\[ \text{(Safe)} \ [ \text{O} \models \text{bad} \iff \text{N := N + 1} \]

\[ \text{(Unfold)} \ [ \text{O} \models \text{bad} \iff \text{N := N + 1} \]

\[ \text{(Safe)} \ [ \text{O} \models \text{bad} \iff \text{N := N + 1} \]

\[ \text{(Unsafe)} \ [ \text{U} \models \text{bad} \iff \text{UNSAFE} \]

**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 \( \{O_i\}_{i \in \mathbb{N}} \), and a must summary \( U \). Intuitively, a query \( (\varphi, i) \) corresponds to checking if \( \varphi \) is reachable for recursion depth \( i \). \( \text{O} \) over-approximates the reachable states for recursion depth \( i \), and \( \text{U} \) under-approximates the reachable states. \( \text{N} \) denotes the current bound on recursion depth. The sequence of may summaries and \( \text{N} \) correspond to the \textit{trace of approximations} and the maximum \textit{level} in IC3/PDR, respectively. For convenience, let \( \text{O}_{i-1} \) be \( \bot \). \( \text{MBP(\varphi, M)} \), for a formula \( \varphi = \text{\tau}_i \varphi_{af} \) and model \( M \models \varphi_{af} \), denotes the result of some MBP function associated with \( \varphi \) for the model \( M \).

Alg. 2 initializes \( N \) to 0 and, \( \text{O}_0 \) and \( \text{U} \) to \( \text{init} \). Candidate initiates a backward search for a counterexample beginning with a set of states in \( \text{bad} \). The potential counterexample is expanded using either \text{DecideMust} or \text{DecideMay}. DecideMust jumps over the call \( \text{Inv}(\tau'') \), in the last CHC of (4), utilizing the must summary \( \text{U} \). DecideMay, on the other hand, creates a query for the call using the may summary of its calling context.
**Successor** updates \( U \) 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 possible to eliminate any index variables in \( \psi \), we strengthen \( \psi \) 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 \( \text{rd}(a, k) < 0 \land \text{rd}(b, k) > 0 \land a \neq b \). However, \( \text{rd}(a, k) = \text{rd}(b, k) \) 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 \( \psi \Rightarrow \neg \phi' \) (as occurs in Conflict). We let \( \psi = \phi_1 \land \phi_2 \) where \( \phi_2 \) 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 \( \phi_2 \) to \( \phi_2' \) such that \( \psi \setminus (\phi_1 \land \phi_2') \) is unsatisfiable and then obtain ITP(\( \psi, \neg(\phi_1 \land \phi_2') \)). In the above example, for \( N = 0 \) we have \( \psi = (a = b) \), \( \phi_1 = (a \neq b) \), and \( \phi_2 = \text{rd}(a, k) < 0 \land \text{rd}(b, k) > 0 \). One can show that \( \phi_2' \) 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
We note that S compares the combined run time
compares S
ervatives of various unwindings of the
formulas corresponding to safety of various unwindings of the
program \[ Z3 \text{ GPDR} \]. We observed that without A
benchmarks. This result confirms our hypothesis.

This shows that S is able to handle only a small fraction of the non-inlined
formulas can grow exponentially. In contrast, the S
framework \[ \text{for the CHC encoding and verification, when inlining is turned}
\text{on and off}. \] A clear advantage is seen in the non-inlining case.

This shows an overwhelming advantage for
S
, 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 S
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

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

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 neces-
sarily finite) MBP procedure, obtaining an instantiation of
the S
framework for verification of safety of sequential
heap-manipulating programs. We show that the new variant of
S
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. Fedryukovich, 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. Grebeshchikov, 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 Check-
ing 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.
\[
\exists a \cdot (\neg (a = t) \land \varphi)
\]

\[
\exists a, j \cdot (rd(a, j) \neq rd(t, j) \land j \notin I \land \varphi)
\]

where \( a \) does not appear in \( t \)

Fig. 9: Modified version of ELIMDISEQ for finite domains.

APPENDIX A
QE AND MBP FOR ARR OVER FINITE INDEX DOMAINS

When finite interpretations of \( I \) are allowed, ELIMDISEQ is no longer an equivalent transformation as there may not exist an index where the arrays in the disequalities disagree on the values. However, one can use extensionality to obtain another equivalent transformation rule ELIMDISEQFINITE, as shown in Fig. 9. As this rule introduces new read terms over \( a \), we need to apply FACTORD once again before ACKERMANN. Also, note that the result of QE and MBP is now of the form \( \exists \vec{x} : I, \vec{v} : V \cdot \psi \).

APPENDIX B
PROOFS OF STATEMENTS ABOUT ARRAYQE AND ARRAYMBP

**Theorem 1:*** ARRAYQE(\( \exists a : arr(I, V) \cdot \varphi \)) returns \( \exists \vec{x} : V \cdot \rho \), where \( \rho \) is quantifier-free and \( \exists \vec{x} : \rho \equiv \exists a : \varphi \).

**Proof:** (Sketch) One can easily show that the rules in Fig. 1, 2, and 3 are equivalence preserving. The theorem follows immediately.

**Theorem 2:** ARRAYQE(\( \exists a \cdot \varphi \)) terminates in time exponential in the size of \( \varphi \).

**Proof:** (Sketch) Line 1 of ARRAYQE essentially eliminates write terms one by one and can be easily shown to terminate. Line 2 can be easily made to terminate by iterating over all partial equality and read terms. The remaining steps of the algorithm clearly terminate as well.

The complexity analysis is similar to that of the decision procedure by Stump et al. [20]. Let \( N \) be the size of \( \varphi \). The number of disjunctions generated by any rewrite rule is bounded by \( N \) (due to the disjunction \( j \in I \) on indices in ELIMWR.EQ). Disjunctions can be generated by the rules for every write term or partial equality and their number is bounded by \( N \). So, the total number of disjunctions generated by the algorithm is bounded by \( O(N^N) \) which is exponential in \( N \). The size of a disjunct generated by a rule can be shown to be bounded by a polynomial in \( N \). CASESPLITEQ can be efficiently implemented using an \((N + 1)\)-way case analysis over all \( N \) partial equalities at once avoiding a Boolean rewriting on line 3 of the algorithm. That is, one can obtain \( N + 1 \) disjuncts, one each for the case of a partial equality being true and the last one for the case of every partial equality being false. Thus, the complexity of ARRAYQE is exponential in \( N \).