

Array Abstractions from Proofs

Ranjit Jhala¹ Kenneth L. McMillan²

¹UC San Diego ²Cadence Berkeley Laboratories

Abstract. We present a technique for using infeasible program paths to automatically infer *Range Predicates* that describe properties of unbounded array segments. First, we build proofs showing the infeasibility of the paths, using axioms that precisely encode the high-level (but informal) rules with which programmers reason about arrays. Next, we mine the proofs for Craig Interpolants which correspond to predicates that refute the particular counterexample path. By embedding the predicate inference technique within a Counterexample-Guided Abstraction-Refinement (CEGAR) loop, we obtain a method for verifying data-sensitive safety properties whose precision is tailored in a program- and property-sensitive manner. Though the axioms used are simple, we show that the method suffices to prove a variety of array-manipulating programs that were previously beyond automatic model checkers.

1 Introduction

Counterexample-guided Abstraction-Refinement(CEGAR)-based techniques [8] have proven to be effective in the verification of control-dominated properties of software [2,15,7,16], chiefly because they precisely track only the small set of facts required to prove the property. However, CEGAR has not had success with data-sensitive properties which require the automatic discovery of abstractions for reasoning about unbounded structures. Consider for example, the following program `init` that initializes an array:

```
for(i=0; i != n; i++) M[i] = 0;
for(j=0; j != n; j++) assert(M[j] == 0);
```

CEGAR-based approaches fail on such programs as for each counterexample path corresponding to an unrolling of k iterations of the upper loop, they infer the atomic predicates: $sel(M, 0) = 0, \dots, sel(M, k - 1) = 0$, which state that the cells 0 through $k - 1$ of the array M have the value 0. These predicates suffice to refute the particular path, but not other, longer paths. Thus, the inability to infer universally quantified predicates about unbounded segments of the array causes CEGAR-based approaches to diverge.

In this paper, we present a technique for using infeasible counterexample paths to infer predicates that describe properties of unbounded array segments and therefore prove many array manipulating programs correct. Our technique is based on two ingredients. The first ingredient is the notion of a *Range Predicate*, an implicitly universally quantified predicate, defined recursively as:

$$RP(t_1, t_2, p) \triangleq p[t_1/\alpha] \wedge (t_1 + 1 = t_2 \vee RP(t_1 + 1, t_2, p))$$

where t_1 and t_2 are terms, respectively called the left and right index, p is an atomic first-order predicate, *i.e.*, an equality, disequality or inequality, which contains an implicitly bound variable α . Intuitively, the range predicate captures the fact that for each element t in the sequence $t_1, t_1 + 1, (t_1 + 1) + 1, \dots$ upto, but not including, t_2 , the fact $p[t/\alpha]$ holds. Thus, the range predicate: $RP(0, i, sel(M, \alpha) = 0)$ states that the first i elements of the array M are equal to 0. Similarly, $RP(0, n, \neg(sel(M, \alpha) \leq 0))$ stipulates that the first n elements of the array M are positive, and $RP(i, n, (sel(M, \alpha) \leq sel(M, \alpha + 1)))$ states that the segment of the array M from index i through n is sorted.

For range predicates to be useful for automatic verification, we require a way to automatically find range predicates relevant to the property being verified. The second ingredient of our technique, is an *axiom-based algorithm* for automatically finding relevant predicates as Craig Interpolants computed from proofs of infeasibility of counterexample paths. We instantiate the algorithm with axioms that precisely encode the high-level, but informal, rules with which programmers reason about arrays, to obtain a method for automatically inferring range predicates tailored to the property to be proved. Thus, the two ingredients are combined to obtain a predicate inference technique which, when embedded within a CEGAR loop [14,18], results in automatic method for verifying data-sensitive safety properties of array-manipulating programs.

To address the challenge of computing range predicate interpolants instead of a divergent sequence of atomic predicates describing individual array cells, our axiom-based algorithm builds upon our previous technique of *L-restricted Interpolation* [17]. Consider the family of languages $L_0 \subseteq L_1 \subseteq \dots$, where L_i is the language of predicates containing numeric constants with absolute value at most i . We set k to 0 and for each candidate counterexample path, try to find an interpolant belonging to L_k . If no such interpolant exists, we increase k and repeat. Thus, if there is an abstraction that suffices to prove the program infeasible, there is some k such that all the predicates of the abstraction belong to L_k , and so the abstraction-refinement loop is guaranteed to terminate. By *restricting* the language we force the solver to find interpolants (and therefore, abstraction predicates) that contain small constants, if these exist. Thus, in the example above, once the counterexample path contains more than $k + 1$ iterations of the first loop, the solver *cannot* return the interpolant $sel(M, 0) = 0, \dots, sel(M, k) = 0$, and is instead forced to find the range predicate $RP(0, i, sel(M, \alpha) = 0)$, which yields an inductive invariant that proves the property.

To compute *L-restricted* interpolants it suffices to find *L-restricted split proofs* where each deduction belongs in L , and where for each deduction, there exists a time step such that the antecedents and consequence of the deduction are over program variables that are in scope at that time step. In Section 3 we show a *local* axiom based algorithm to generate split proofs of refutation, and therefore, interpolants. In Section 4 we present an instantiation of this framework using range predicate axioms. Though the axioms for reasoning about range predicates are simple, we present initial experiments in Section 5 that indicate

that they are expressive enough to efficiently capture a variety of idiomatic uses of arrays, in a manner that is precise enough to prove data intensive properties.

Related Work. The problem of synthesizing abstractions and invariants for arrays and other unbounded data structures has received much attention. One line of research uses *templates* representing families of candidate loop invariants (*e.g.* affine constraints over program variables) to generate loop invariants [3,4,29]. These approaches use a template of quantified invariants derived from [6], where the problem of *checking* a given quantified invariant is studied. A second line of work uses abstract interpretation based techniques for shape analysis. Examples include those based on three-valued logic [28,22] and Separation Logic [10]. The abstract domain for arrays presented in [12] captures properties similar to range predicates. Predicate abstraction [13] based approaches for shape analysis [11,9,5,26,21,1] can also be viewed as an instance of abstract interpretation. In the approaches which work for unbounded structures an expert must supply appropriate predicates or instrumentation predicates which are combined via a fixpoint computation to obtain an inductive invariant. Several authors have proposed using specialized rules to build decision procedures [26], and more generally, program analyses [27].

2 Overview

We begin with an overview of safety verification via interpolant-based abstraction refinement.

Notation. In this paper, we use standard first-order logic (FOL). By $\mathcal{L}(\Sigma)$ we refer to the set of well-formed formulas over a vocabulary Σ of non-logical symbols, and for a given formula ϕ we use $\mathcal{L}(\phi)$ to denote the set of well-formed formulas over the vocabulary of non-logical symbols occurring in ϕ .

We assume that for every (non-logical) symbol s , there exists a unique symbol s' , *i.e.*, s with one prime added. We use s with n primes added to represent the value of s at time step n . For any formula or term ϕ , we write $\phi^{(n)}$ to denote the addition of n primes to every non-logical symbol in ϕ . Finally, for any set of symbols Σ , we write $\Sigma^{(n)}$ to denote $\{s^{(n)} \mid s \in \Sigma\}$ and Σ' to denote $\Sigma^{(1)}$. For a term (resp. predicate) t we write $t[e/x]$ to denote the term (resp. predicate) obtained by substituting all occurrences of x in t with e .

Programs. We model programs abstractly using logical formulas, and restrict ourselves to single-procedure programs. Let S be a set of *state variables* corresponding to individual program variables. A *state formula* is a formula in $\mathcal{L}(S)$, which may contain interpreted symbols like $+$, $=$, \leq in addition to the symbols in S . A *transition* is a formula in $\mathcal{L}(S \cup S')$. A *program* is a pair (\mathcal{T}, Π) where \mathcal{T} is a finite set of transitions and $\Pi \subseteq \mathcal{T}^*$ is a regular language of sequences of transitions. Intuitively, each command (basic block or branch condition) of the control-flow graph of the program corresponds to a transition, and the set of paths is the regular set of *syntactic* control-flow paths of the program.

The following are the transitions for the program `init` from Section 1. For each transition, we omit for clarity the implicit conjunction $x = x'$ for all pro-

gram variables x whose primed version is not explicitly shown in the transition.

$$\begin{aligned} T_1 : i' = 0 & & T_2 : i \neq n \wedge M' = \text{upd}(M, i, 0) \wedge i' = i + 1 & & T_3 : i \neq n \\ T_4 : j' = 0 & & T_5 : j \neq n \wedge \text{sel}(M, j) = 0 \wedge j' = j + 1 & & T_6 : j \neq n \wedge \text{sel}(M, j) \neq 0 \end{aligned}$$

The set of syntactic paths of the program that lead to the **assert** failure are given by the regular expression: $T_1 \cdot T_2^* \cdot T_3 \cdot T_4 \cdot T_5^* \cdot T_6$.

Path Constraints. A *path* π is a sequence of transitions T_0, \dots, T_n in Π . For any path π , the *constraints* $\text{Cons}(\pi)$ is the sequence of formulas $T_0^{(0)}, \dots, T_n^{(n)}$. A path π is *infeasible* if the *path formula* $\bigwedge \text{Cons}(\pi)$ is inconsistent, *i.e.*, unsatisfiable. A program (\mathcal{T}, Π) is *infeasible* if every path in Π is infeasible.

The path formula represents all possible concrete program executions that follow the given control-flow path. A satisfying assignment for the path formula can be mapped back to the values taken by the program variables at each time step from 0 (the initial value) through $n + 1$ (at the end of the path). Thus, if the formula is satisfiable, the path corresponds to a feasible concrete execution of the program. The left side of Figure 1 shows the path constraints corresponding to the path where the upper and lower loops are unrolled twice. From top to bottom, the constraints shown are the formulas: $T_1^{(0)}, T_2^{(1)}, T_2^{(2)}, T_3^{(3)}, T_4^{(4)}, T_5^{(5)}, T_6^{(6)}$. Using the standard axioms for equality and arithmetic, and McCarthy’s axioms for *sel* and *upd*, one can check that the path formula shown on the left in Figure 1 is inconsistent.

Safety Verification. Informally, the safety verification problem is to determine whether the program always avoids entering a set of undesirable “error” states. We can reduce the safety verification problem to that of determining if a program is infeasible, by intersecting Π with the set of paths leading to the “error” states.

Interpolants. For a sequence of formulas $\Gamma = A_0, \dots, A_n$, we say that $\hat{\Gamma} = \hat{A}_0, \dots, \hat{A}_{n+1}$ is an *interpolant* for Γ if: (1) $\hat{A}_0 = \text{TRUE}$ and $\hat{A}_{n+1} = \text{FALSE}$, and, (2) for all $0 \leq i \leq n$, $\hat{A}_i \wedge A_i$ implies \hat{A}_{i+1} , and, (3) for all $0 \leq i \leq n$, $\hat{A}_{i+1} \in \mathcal{L}(A_i) \cap \mathcal{L}(A_{i+1})$

Interpolants are Abstractions. For any infeasible path, the sequence of formulas of the interpolant for the path constraints *overapproximate* the possible program configurations along the path in a manner that is *precise* enough to demonstrate the infeasibility of the path. To see this, observe that the interpolant for path formula $T_0^{(0)}, \dots, T_n^{(n)}$ is a sequence of formulas $\hat{T}_0, \dots, \hat{T}_{n+1}$, such that: (1) \hat{T}_0 is **TRUE**, representing all possible initial states and \hat{T}_n is **FALSE**, indicating that there is no possible state at the end of the path, (2) for all $0 \leq i \leq n$, executing the transition T_i from a state in \hat{T}_i takes the system into a state in \hat{T}_{i+1} , and, (3) for all $0 \leq i \leq n$, the set of possible states for time i is expressed as a state formula over the values of the variables at time i .

Thus, the interpolant corresponding to an infeasible path can be used to iteratively refine an abstract model of the program either directly [24], or indirectly by predicate abstraction over the set of atomic predicates appearing in the interpolant [14]. This process is repeated until all paths are shown infeasible or a feasible path is found [8]. For example, for the path constraints shown in Figure 1, a possible interpolant is the sequence of formulas: **TRUE**, $(i^{(1)} = 0)$,

Path Constraints	Interpolant
	TRUE
$T_1^{(0)} : i^{(1)} = 0$	$i^{(1)} = 0$
$T_2^{(1)} : i^{(1)} \neq n \wedge M^{(2)} = \text{upd}(M^{(1)}, i^{(1)}, 0) \wedge i^{(2)} = i^{(1)} + 1$	$RP(0, i^{(2)}, \text{sel}(M^{(2)}, \alpha) = 0)$
$T_2^{(2)} : i^{(2)} \neq n \wedge M^{(3)} = \text{upd}(M^{(2)}, i^{(2)}, 0) \wedge i^{(3)} = i^{(2)} + 1$	$RP(0, i^{(3)}, \text{sel}(M^{(3)}, \alpha) = 0)$
$T_3^{(3)} : i^{(3)} = n$	$RP(0, n, \text{sel}(M^{(4)}, \alpha) = 0)$
$T_4^{(4)} : j^{(5)} = 0$	$RP(j^{(5)}, n, \text{sel}(M^{(5)}, \alpha) = 0)$
$T_5^{(5)} : j^{(5)} \neq n \wedge \text{sel}(M^{(5)}, j^{(5)}) = 0 \wedge j^{(6)} = j^{(5)} + 1$	$RP(j^{(6)}, n, \text{sel}(M^{(6)}, \alpha) = 0)$
$T_6^{(6)} : j^{(6)} \neq n \wedge \text{sel}(M^{(6)}, j^{(6)}) \neq 0$	FALSE

Fig. 1. On the left, we show the path constraints generated by the path leading to the assertion violation in `init` where each of the loops is unrolled twice. For $i \geq 3$ the i -th path constraint has an additional conjunct $M^{(i+1)} = M^{(i)}$ omitted for brevity. The right column shows the interpolants generated using range predicates. We write the $i + 1$ -th element of the interpolant sequence to the right of the i -th path constraint. Note that the $i + 1$ -th element of the interpolant sequence is implied by the conjunction of the i -th element and the i -th path constraint.

$(i^{(2)} = 1), (\text{sel}(M^{(3)}, 1) = 0), (\text{sel}(M^{(4)}, 1) = 0), (\text{sel}(M^{(5)}, 1) = 0 \wedge j^{(5)} = 0),$
 $(\text{sel}(M^{(6)}, 1) = 0 \wedge j^{(6)} = 1), \text{FALSE}$. After dropping the superscripts, we get a set of predicates: $i = 0, i = 1, \text{sel}(M, 1) = 0, j = 0, j = 1$, that suffices to refute paths where the upper loop is unrolled at most two times.

(In)Completeness. Even though the atomic predicates suffice to eliminate the given path, more predicates may be needed for longer paths, *e.g.* those corresponding to more iterations through the loop. In our example, each path corresponding to j iterations of the upper loop will result in new predicates constraining the first j elements of the array to be zero, but which are insufficient to refute longer paths. As a result, the iterative abstraction-refinement diverges.

Range Predicates. We obtain the interpolant sequence shown on the right in Figure 1, by giving the interpolating procedure *axioms* for reasoning about range predicates and simultaneously restricting it to find interpolants in the language L_1 (using numeric constants of absolute value at most 1). Note that the restriction forces the solver to return an interpolant that states that all cells from 0 through n have been initialized with zero for the point after the first loop has finished. After dropping the superscripts, we obtain the set of new abstraction predicates: $i = 0, RP(0, i, \text{sel}(M, \alpha) = 0), RP(0, n, \text{sel}(M, \alpha) = 0), RP(j, n, \text{sel}(M, \alpha) = 0)$. Thus, perhaps counter-intuitively [4], from a finite path we can deduce predicates describing unbounded array segments, simply by restricting the language of the interpolants. Subsequent predicate abstraction over these predicates refutes this particular path and in fact, results in an inductive invariant that proves the program infeasible.

3 Generating Interpolants from Axioms

We now consider the problem of using a specialized set of axioms (in addition to the axioms belonging to the ground theories of equality, uninterpreted functions and difference constraints) to find L -restricted interpolants for a given sequence of formulas $\Gamma = A_0, \dots, A_n$.

As shown in [17], this can be achieved by the following two-step process. First, we must find an L -restricted *split proof* where each deduction can be mapped to a time step i such that the antecedents and consequence of the deduction belong to $\mathcal{L}(A_i)$, and if there are no antecedents, the consequence is implied by A_i . Second, we can convert the split proof into a set of propositional clauses (by converting each atom into a literal) and then use propositional interpolation [23] to find an interpolant. The latter operation is polynomial in the size of the split-proof and results in interpolants whose atoms appear in the split proof and are thus from the restriction language L .

Split Proofs. An L -restricted *split proof* over a set of hypotheses $\Gamma = A_0, \dots, A_n$ is a triple (V, E, N) , where V is a set of formulas, (V, E) is a directed acyclic graph, and N is a labeling function from V to $[0 \dots n]$ such that:

- for each vertex $v \in V$, we have $A_{N(v)}, \{u \mid (u, v) \in E\} \models v$, and,
- for each edge $(u, v) \in E$, we have $u, v \in \mathcal{L}(A_{N(v)})$, and,
- for each edge $(u, v) \in E$, if $N(u) \neq N(v)$ then $u \in L$.

A L -restricted *split refutation* of Γ is an L -restricted split proof over Γ whose unique sink vertex (no out-edges) is FALSE.

Intuitively, a split proof is one where as before, each deduced formula (vertex) can be localized to a particular time step (the formula’s label) – the formula is implied by the conjunction of previously deduced facts (the vertex’s predecessors) and the hypotheses corresponding to the formula’s time step. Moreover, if a formula is deduced at a time step different from those at which a predecessor was deduced, then the predecessor formula must belong to the restriction language L . In other words, within a time step (*e.g.* within the constraints corresponding to a large basic block of code), we may deduce formulas not in the restriction language L , as these formulas will not appear in the subsequent interpolant.

We call a sequence of hypotheses $\Gamma = A_0, \dots, A_n$ *strict* if for all i, j such that $|i - j| > 1$ we have $\mathcal{L}(A_i) \cap \mathcal{L}(A_j) = \emptyset$. It is easy to check that the sequence of hypotheses corresponding to path constraints are strict.

Theorem 1. [17] *Given a strict sequence of hypotheses Γ and a propositionally closed language L , Γ has an L -restricted interpolant iff it has a L -restricted split refutation.*

Generating Proofs from Local Axioms

Thus, to find L -restricted interpolants for Γ , we need to find L -restricted split refutations of Γ . The problem of generating L -restricted split refutations for

```

1: Input: Local axioms  $A$ 
2: Input: Hypotheses  $\Gamma = A_0, \dots, A_n$ 
3: Output: A split refutation of  $\Gamma$ 
4: indexed := Seed( $\Gamma$ )
5: while FALSE  $\notin$  pf.V do
6:   choose  $a, I, j$  from  $A$ , indexed,  $[0, \dots, n]$ 
7:   match Project( $a, I, j$ ) with Some ( $I', Q', c'$ )
8:     if  $c' \notin$  pf.V and  $\forall q \in Q'. \text{Query}(q)$  then
9:       pf.V := pf.V  $\cup$   $\{c'\}$ 
10:      pf.E := pf.E  $\cup$   $\{(h, c') \mid h \in I' \cup Q'\}$ 
11:      pf.N( $c'$ ) :=  $j$ 
12:      indexed := indexed  $\cup$   $c'$ 
13:      Assert( $c'$ )
14: return Cone(pf, FALSE)

```

Fig. 2. Procedure Generate

Program	Time	Preds	Iter
init	1.190	18	7
vararg	1.520	14	8
copy	3.650	20	11
copy-prop	9.720	38	17
find	2.240	20	12
partition	7.960	37	14
part-init	4.630	32	12
producer	45.000	39	41
insert	91.220	74	36
scull	9.180	36	14

Fig. 3. Experimental Results: **Time** is the total number of seconds spent to prove the program safe, **Preds** is the number of predicates required, **Iter** is the number of iterations of the abstraction-refinement loop, Experiments were run on an IBM T42 Laptop with a 1.7GHz processor and 512Mb RAM.

formulas over theories of equality, uninterpreted functions, difference bounds and restricted use of the array operators “*sel*” and “*upd*” was addressed in [17]. Thus, we assume there is a “ground” procedure that handles the above theories and describe how this procedure can be *extended* with specialized axioms.

Local Axioms. A *local axiom* a is a partial function that takes as input a set of *index* formulas I and returns a pair of *query* formulas Q and a *consequence* formula c , such that (1) $I, Q \models c$, and, (2) $Q, c \in \mathcal{L}(I)$. Intuitively, for a given set of index formulas that is known to be true, there is a unique set of query formulas over the ground theory which if additionally true imply the consequence formula. To ensure that axiom instantiation results in split proofs, we require that the queries and consequence belong to the same language as the index formulas.

Algorithm Generate. Our non-deterministic algorithm **Generate** for finding split refutations for a sequence of hypotheses $\Gamma = A_0, \dots, A_n$ is shown in Figure 2. The algorithm takes as input a set of local axioms A and a sequence of hypotheses Γ . It maintains a set of index formulas in the variable indexed, and a split proof pf whose vertices correspond to all the facts that have been deduced. The overall structure of the algorithm is similar to that of saturation-based provers [20]. First (line 4), it seeds the set of indexed formulas using the formulas that the ground procedure derives from Γ . Next, (lines 5–13) it goes into a loop where it repeatedly selects a set of index formulas and a candidate axiom and attempts to derive new facts by applying the axiom to the index formulas, until a contradiction is found (*i.e.*, FALSE is deduced).

Project. The main challenge in our setting is that we need to ensure that all the deductions can be *localized to a single time step* — we must ensure that whenever we deduce a consequence c from hypotheses I, Q , there must be some time step $j \in 0 \dots n$ such that I, Q, c belong in $\mathcal{L}(A_j)$, *i.e.*, contain symbols that are local to time step j . Note that the local axiom functions are defined only on index formulas belonging to some common time step. Thus, to make the procedure `Generate` complete, we would have to undertake the expensive task of maintaining different *representatives* for each formulas at each time step at which the formula has a congruent version. We avoid this by using a procedure `Project` that takes as input a set of formulas, possibly belonging to languages of different time steps, an axiom, and a target time step j and determines whether there is a set of formulas I' at time step j such that: (1) for each formula in I , there is a congruent version in I' , and, (2) each formula in I' belongs to $\mathcal{L}(A_j)$, and, (3) the application of the axiom to the index formulas I' yields the query formulas Q' and a consequence c' . If a suitable I' exists, the procedure updates the split proof `pf` with congruence proofs (using the axioms for equality and uninterpreted functions) for the elements of I' , and returns the tuple of (I', Q', c') .

We use the `Project` function in the main loop as follows. In each iteration we choose an axiom `a`, a set of index formulas (from arbitrary time frames) I , and a target time frame j (line 6), and we call `Project` to determine if at time j there is a congruent version I' with query formulas Q' and consequence c' , all of which belong to time step j . If `Project` succeeds (line 7), we check if the consequence c' is not a previously known fact, and invoke the ground procedure `Query` to determine whether each of the queries in Q' is true (line 8). For each provable query, the ground procedure updates the split proof `pf` with vertices for the query formula. If all the queries Q' are provable and the consequence c' is new, the split proof `pf` is updated with the new consequence (lines 9–12) and the consequence is asserted to the ground procedure (line 13). If this assertion yields a contradiction *i.e.*, causes the ground procedure to deduce `FALSE`, the algorithm returns a split refutation which is the backwards transitive closure of `FALSE` in the split proof `pf`.

Correctness and Termination. When the procedure `Generate` finishes, it returns a split refutation for the hypotheses I . The presented procedure is abstracted for clarity. In practice, by ensuring that we iterate over the indexed formulas exhaustively we can guarantee that the procedure will find a split refutation if one exists. The procedure can be terminated when it reaches a point where no new facts in the restriction language can be deduced. Termination follows as we restrict the language to bound the set of candidate formulas.

4 Axioms for Range Predicates

We now describe an instantiation of the framework of the previous section with axioms for reasoning about *Range Predicates* which describe properties of contiguous blocks of array elements. As range predicates capture facts that hold

for *sequences* of array indices, we devise axioms that: *generalize* range predicates from facts that hold about a single index, *instantiate* range predicates to individual indices, *extend* range predicates to longer sequences, *shrink* range predicates to shorter sequences, *join* range predicates over “adjacent” sequences, and, *preserve* range predicates in the presence of array updates.

Figure 4 shows a representative subset of the axioms used for reasoning about range predicates. We use η as an abbreviation for the map $\lambda t.t + 1$ from terms to terms. Each axiom is shown as a proof rule, with the antecedents above the line and the consequence below the line. The antecedents within boxes are the index formulas, and those not in boxes are query formulas. Due to lack of space, we omit the meta-theorems that show the soundness of the range predicate axioms with respect to the recursive range predicate definition, and therefore show that the axioms only yield semantically valid derivations.

Generalize: The axiom **Generalize** for creating range predicates simply takes an ordinary formula and replaces occurrences of terms t inside the formula with α , to obtain a range predicate that holds from t to $\eta(t)$.

Instantiate: There are two rules for instantiating a range predicate: **Instantiate-Left** for instantiating with the left index, and **Instantiate-Right** for the right index. In either case, the consequence is p with α substituted with the appropriate index.

Extend: There are two rules for extending a range predicate: **Extend-Left** for extending at the left end and **Extend-Right** for extending at the right end. In either case, the axiom has an antecedent query formula that the predicate p hold at the appropriate index.

Shrink: There are two rules for shrinking a range predicate: **Shrink-Left** for shrinking at the left end and **Shrink-Right** for shrinking at the right end. In either case, the axiom has an antecedent query formula that ensures that in the result, the left and right indices are disequal. This ensures the soundness of the instantiation axioms.

Preserve: The trickiest rules are those that ensure that an update to the array preserves the properties captured by a given range predicate *i.e.*, the properties continue to hold in the updated array, as long as the update happens “outside” the range of indices of the range predicate. Both rules require a syntactic condition that the read address t be *linear*, *i.e.*, that α not appear under any function symbol inside t . The **Preserve-Right** rule states that for any linear read address t parameterized by α , if the address obtained by substituting α with the right index is less than (*i.e.*, to the left of) the address written to (a), then the reads return the same values in the updated array as the update does not affect the addresses read through t . The **Preserve-Left** rule is the symmetric version for writes to the left of the left index of the range predicate.

Join: The rule **Join** is used to join two adjacent range predicates.

Example Split Proof. Figure 5 shows a split proof generated by FOCI , to refute the path constraints from Figure 1. The constraints are simplified using

$$\begin{array}{c}
\frac{\boxed{p}}{RP(t, \eta(t), p[\alpha/t])} \text{ Generalize} \frac{\boxed{RP(t_1, t_2, p)} \quad \boxed{RP(t_2, t_3, p)}}{RP(t_1, t_3, p)} \text{ Join} \\
\frac{\boxed{RP(t_1, t_2, p)}}{p[t_1/\alpha]} \text{ Instantiate-Left} \frac{\boxed{RP(t_1, \eta(t_2), p)}}{p[t_2/\alpha]} \text{ Instantiate-Right} \\
\frac{\boxed{RP(\eta(t_1), t_2, p)} \quad p[t_1/\alpha]}{RP(t_1, t_2, p)} \text{ Extend-Left} \frac{\boxed{RP(t_1, t_2, p)} \quad p[t_2/\alpha]}{RP(t_1, \eta(t_2), p)} \text{ Extend-Right} \\
\frac{\boxed{RP(t_1, t_2, p)} \quad \eta(t_1) \neq t_2}{RP(\eta(t_1), t_2, p)} \text{ Shrink-Left} \frac{\boxed{RP(t_1, \eta(t_2), p)} \quad t_1 \neq t_2}{RP(t_1, t_2, p)} \text{ Shrink-Right} \\
\frac{\boxed{RP(\eta(t_1), t_2, p)} \quad a \leq t[t_1/\alpha] \quad t \text{ is linear}}{RP(\eta(t_1), t_2, p[\text{sel}(\text{upd}(M, a, v), t)/\text{sel}(M, t)])} \text{ Preserve-Left} \\
\frac{\boxed{RP(t_1, t_2, p)} \quad t[t_2/\alpha] \leq a \quad t \text{ is linear}}{RP(t_1, t_2, p[\text{sel}(\text{upd}(M, a, v), t)/\text{sel}(M, t)])} \text{ Preserve-Right}
\end{array}$$

Fig. 4. Axioms for Reasoning about Range Predicates

Static Single Assignment form [14], which avoids the equality constraints that “copy” unmodified variables across a transition. In particular, we omit the copy constraints for M after time step 3 (as the array is not updated subsequently). For brevity, we write x^i for $x^{(i)}$. On the left side we show the formulas corresponding to the split proof vertices on the right. Each square vertex is a hypothesis labeled by the time step to which the hypothesis belongs (*i.e.*, a hypothesis in A_j is labeled j). Each circular vertex is a deduction, made using the range predicate axioms or the axioms for equality, congruence or arithmetic, labeled by the time step at which the deduction was made. The curly braces describe how sub-proofs were generated, either by the application of a range predicate axiom (formulas at lines 1,7,9,20,22), or via **Project** which uses one or more applications of the axiom of congruence (formulas at lines 5,12,14,16,19). Notice that the formula at line 9 contains variables that do not belong in time step 5 where the **Shrink-Left** axiom can be applied, but which is congruent, and therefore is **Project**-ed to the formula at line 16 which does belong at time step 5.

5 Experiences

We now describe our experiences so far with implementing the technique and applying it to verify programs, some lessons drawn from our experiments and some possible avenues for future work.

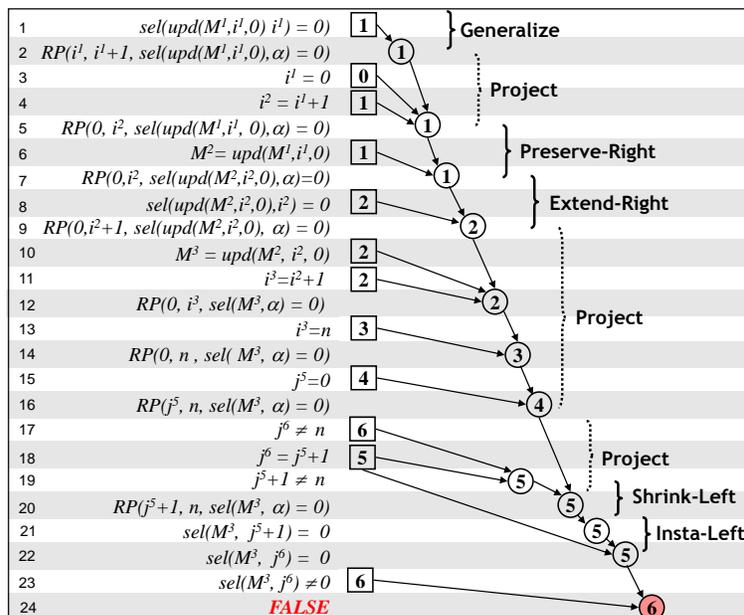


Fig. 5. Split proof generated by FOCI for the path constraints from Figure 1.

Implementation. We have extended the FOCI split prover [17] with axioms for range predicates. Our current implementation is specialized to the axioms for range predicates and has an overall structure similar to procedure `Generate` from Section 3, but with a few differences. First, instead of applying the generalizing rule to all the atomic predicates deduced by the ground procedure, we only generalize from a set of predicates that are obtainable by some syntactic manipulation of the input constraints. Second, there are heuristics to bias the prover to find simpler proofs, with more general interpolants, *i.e.*, those which are less specific to the particular path whose constraints are fed to the prover. The extended prover is integrated with BLAST [14]. As the predicates found require disjunctive images, we use FOCI to iteratively refine the transition relation using the method presented in [18].

Experiments. In preliminary experiments, we have applied the model checker extended with range predicates to a variety of small array-intensive programs hitherto beyond the grasp of automatic refinement based tools. The results are summarized in Table 3. `init` is the example from Section 2. `vararg` is an instance of the common idiom in C programs for scanning the buffer of arguments to determine how many input parameters were passed into the program, by repeatedly increasing an index until a NULL cell is found, and then going backwards

(decreasing the counter) dereferencing the contents of the array to extract the arguments. The property proved is that in the second phase, the values dereferenced are non-NULL. `copy` (from [12]) simply copies the contents of one array into another, and then asserts that the two are the same. `copy-prop` first checks that a given source array has only non-zero elements, then copies the array into a destination and asserts that the destination has only non-zero elements. `find` (from [11]) scans the array trying to find the index at which a particular value resides, and returns -1 if the value is not in the array. The property checks that if -1 is returned then the value is not in the array. `partition` (from [4]) copies the zero and non-zero elements of a source array into two different arrays and checks that the two destination arrays only have zeros and non-zeros respectively. `part-init` (from [12]) copies those indices of a source array for which the source array's value is positive, into a target array, and checks that at the index stored in the target array the source array's value is indeed positive. `producer` is a producer-consumer example, where a producer keeps generating a sequence of values and writes them into increasingly larger array indices via a `head` index that is incremented, while a consumer consumer uses a `tail` index to read the values stored in the array. The property checked is that the sequence of values written by the producer is the same as those read by the consumer. `insert` is an in-place insertion routine, that takes a sorted array and inserts a new element into the appropriate place by repeatedly swapping elements until the right position is found (*i.e.*, the inner loop in an insertion sort procedure). The property checked is that after the insertion, the extended array is sorted. `scull` is a text-book Linux device driver for which we check a property that requires an array of devices to be appropriately initialized.

Discussion and Future Work. Our experiments show that the axioms are expressive enough to capture many of the idiomatic uses of arrays, while yielding property-sensitive abstractions. However, there are several deficiencies in the approach that need to be remedied with future work. The main difficulty with the approach is that the prover may find proofs which refute short paths, but which do not generalize to longer paths, thereby delaying convergence, or worse, cluttering the abstraction with irrelevant facts about the program causing image computation to explode. This problem arose in our (as yet unsuccessful) attempt to prove that an implementation of `insertion-sort` correctly sorted an array. Though the range predicate axioms suffice to prove the property, BLAST is overwhelmed by irrelevant predicates generated by smaller paths. Thus, one possible line to pursue is to find ways to make the outer loop converge more rapidly. Finally, we view this work as first step towards an axiom-extensible technique for verifying data-sensitive properties. To this end, we would like to implement a generalized split-proof engine parameterized by axioms, and devise and instantiate it with axioms for other data structures like lists [26,21], hash tables and richer logical constructs like separating conjunctions [10].

Acknowledgements. We would like to thank Cormac Flanagan, Alan Hu, Rupak Majumdar, Zvonimir Rakamaric, Tom Reps, Andrey Rybalchenko, and the

anonymous referees for extremely useful comments about a previous version of the paper. The first author was supported in part, by NSF grant CNS-0541606.

References

1. Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis by predicate abstraction. In *VMCAI*, pages 164–180, 2005.
2. T. Ball and S.K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.
3. Dirk Beyer, Thomas Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis in combination theories. In *VMCAI*, 2007.
4. Dirk Beyer, Thomas Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *PLDI*, 2007 (to appear).
5. Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin Rinard. Using first-order theorem provers in the Jahob data structure verification system. In *Verification, Model Checking and Abstract Interpretation*, November 2007.
6. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What’s decidable about arrays? In *VMCAI*, pages 427–442, 2006.
7. S. Chaki, E.M. Clarke, A. Groce, and O. Strichman. Predicate abstraction with minimum predicates. In *CHARME 03*, LNCS 2860, pages 19–34. Springer, 2003.
8. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00.*, LNCS 1855, pages 154–169. Springer, 2000.
9. Dennis Dams and Kedar S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *VMCAI*, pages 310–324, 2003.
10. Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.
11. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL 02: Principles of Programming Languages*, pages 191–202. ACM, 2002.
12. Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.
13. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer Aided Verification*, LNCS 1254, pages 72–83. Springer, 1997.
14. T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan. Abstractions from proofs. In *POPL 04*, pages 232–244. ACM, 2004.
15. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL 02: Principles of Programming Languages*, pages 58–70. ACM, 2002.
16. Himanshu Jain, Franjo Ivancic, Aarti Gupta, and Malay K. Ganai. Localization and register sharing for predicate abstraction. In *TACAS*, pages 397–412, 2005.
17. R. Jhala and K.L. McMillan. A practical and complete approach to predicate refinement. In *TACAS 06*, LNCS 2987, pages 298–312. Springer-Verlag, 2006.
18. Ranjit Jhala and Kenneth L. McMillan. Interpolant-based transition relation approximation. In *CAV 05*, LNCS, pages 39–51. Springer, 2005.
19. Daniel Kroening and Natasha Sharygina. Approximating predicate images for bit-vector logic. In *TACAS*, pages 242–256, 2006.
20. Shuvendu K. Lahiri, Thomas Ball, and Byron Cook. Predicate abstraction via symbolic decision procedures. In *CAV*, pages 24–38, 2005.
21. Shuvendu K. Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In *POPL*, pages 115–126, 2006.
22. T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *SAS 00: Static Analysis Symposium*, LNCS 1824. Springer, 2000.

23. Kenneth L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.
24. Kenneth L. McMillan. Lazy abstraction with interpolants. In T. Ball and R. Jones, editors, *CAV 2006: Computer-Aided Verification*, LNCS, pages 123–136. Springer-Verlag, 2006.
25. K.L. McMillan. Interpolation and SAT-based model checking. In *CAV 03: Computer-aided Verification*, Lecture Notes in Computer Science 2725, pages 1–13. Springer, 2003.
26. Zvonimir Rakamaric, Jesse D. Bingham, , and Alan J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *VMCAI*, page (to appear), 2007.
27. Thomas W. Reps. Demand interprocedural program analysis using logic databases. In *Workshop on Programming with Logic Databases (Book), ILPS*, pages 163–196, 1993.
28. Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
29. Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, pages 25–41, 2005.