On Solving Universally Quantified Horn Clauses

Nikolaj Bjørner[1], Ken McMillan[1], and Andrey Rybalchenko[1,2]

[1] Microsoft Research
[2] Technische Universität München

Abstract. Program proving can be viewed as solving for unknown relations (such as loop invariants, procedure summaries and so on) that occur in the logical verification conditions of a program, such that the verification conditions are valid. Generic logical tools exist that can solve such problems modulo certain background theories, and therefore can be used for program analysis. Here, we extend these techniques to solve for *quantified* relations. This makes it possible to guide the solver by constraining the form of the proof, allowing it to converge when it otherwise would not. We show how to simulate existing abstract domains in this way, without having to directly implement program analyses or make certain heuristic choices, such as the terms and predicates that form the parameters of the abstract domain. Moreover, the approach gives the flexibility to go beyond these domains and experiment quickly with various invariant forms.

1 Introduction

Many problems of inference in program verification can be reduced to relational post-fixed point solving [13, 4]. That is, to prove correctness of a program, we apply the rules of a program proof system to obtain purely logical proof subgoals. These subgoals are called the *verification conditions* or VC's. Proving validity of the VC's implies correctness of the program. The VC's generally contain auxiliary predicates such as inductive invariants, that must be *inferred*. Leaving these auxiliary predicates undefined (that is, as symbolic constants) the problem of inference becomes a problem of *solving* for unknown relations satisfying a set of logical constraints. In the simplest case, this is an SMT (satisfiability modulo theories) problem.

```
procedure π(ref a : int array, N : int):
    var i : int := 0;
    while i < N invariant P(a, i, N) do
        a[i] := 0;
        i := i + 1;
    done
    assert ∀0 ≤ j < N. a[j] = 0;
```

Consider, for example, the simple procedure on the left. The procedure sets elements $0 \ldots N-1$ of array a to zero in a loop. We want to prove that on return, these array elements are in fact zero (leaving aside the question of array over-run). The loop is annotated with an invariant $P(a, i, N)$, an unknown predicate which we wish to discover in order to prove correctness.

The *verification conditions* of this program are the following three logical formulas:

$$i = 0 \implies P(a, i, N) \tag{1}$$

$$P(a, i, N) \wedge i < N \implies P(a[i \leftarrow 0], i + 1, N) \tag{2}$$

$$P(a, i, N) \wedge \neg(i < N) \implies \forall 0 \leq j < N.\ a[i] = 0 \tag{3}$$

These are just the three proof obligations of the Hoare logic rule for while loops translated into logic. They say, respectively, that invariant $P(a, i, N)$ is initially true, that it is preserved by an iteration of the loop, and that it implies the post-condition of the loop on exit. Note in particular that $P(a[i \leftarrow 0], i + 1, N)$ is just the weakest precondition of $P(a, i, N)$ with respect to the loop body. The notation $a[i \leftarrow 0]$ means "array a with index i updated to 0".

The problem of proving the program thus reduces to finding an interpretation of the unknown predicate (or relation) P that makes the VC's valid. In the simplest case (technically, when the background theory is complete) a program proof is a satisfying assignment for the VC's. In our example, one solution is:

$$P(a, i, N) \equiv \forall 0 \leq j < i.\ a[j] = 0.$$

Of course this does not mean that we can in practice solve the VC's using existing SMT solvers. The VC's are valid when they are true with the free variables a, i, N universally quantified. SMT solvers generally cannot produce models for arbitrary quantified formulas.

There are specialized solvers, however, that can take advantage of the fact that the VC's fit a particular pattern. We will use the notation $\phi[X]$ to stand for a formula or term with free variables X. Our VC's have the form $\forall X.B[X] \implies H[X]$. We will call $B[X]$ the *body* of the VC and $H[X]$ the head. Our unknown relations such as P occur only in a limited way in these formulas: as the head, or as a conjunct of the body. Thus, supposing P, Q are our unknown relations, we could have $\forall x.\ x < 0 \wedge P(x) \implies Q(x)$ or $\forall x, y.\ P(x) \wedge Q(x, y) \implies x < y$ as VC's, but not $\forall x, y.\ x < y \implies P(x) \vee Q(x)$. Another way to say this is that our VC's are *constrained Horn clauses*.

Alternatively, we can view the VC's as a Constraint Logic Program (CLP) [20]. In our example, (1) and (2) are the clauses of the program. The VC (3) can be viewed as a safety property to be proved of the program, or its negation can be considered a *goal*, an *answer* to which would be a counterexample to program correctness. In fact, numerous schemes for capturing program semantics in CLP have been proposed [7, 10, 22].

A number of tools exist for solving constrained Horn clauses modulo different theories, or equivalently proving CLP programs. For example, QARMC can solve such problems modulo rational linear arithmetic. It uses a technique based on predicate abstraction [12] and Craig interpolation [26] that was generalized to trees [15]. Bjørner and Hoder describe a technique for the same theory that is an extension of property-driven reachability analysis [17]. A system called Duality [30] uses an extension of the IMPACT algorithm [27] to solve problems in

$$i = 0 \implies \forall j.\ P(a[j], i) \tag{4}$$
$$\forall j.\ P(a[j], i) \wedge i < N \implies \forall j.\ P(a[i \leftarrow 0][j], i + 1) \tag{5}$$
$$\forall j.\ P(a[j], i) \wedge \neg(i < N) \implies \forall 0 \leq j < N.\ a[j] = 0 \tag{6}$$

Fig. 1. VC's generated for array initialization, with quantified invariant.

the combined theory of integer linear arithmetic and arrays, using an interpolating decision procedure for these theories [29]. Likewise, the Eldarica tool [32, 19] takes as input constrained Horn clauses over Presburger arithmetic. The Spacer tool [24] combines CEGAR (counter example guided abstraction refinement) with 2BMC (bounded model checking based model checking) for solving Horn clauses.

Tools based on CLP include CiaoPP [16] which takes an abstract interpretation approach and TRACER [21] that uses a hybrid of symbolic execution, interpolation and abstract interpretation.

The advantage of such solvers is that they abstract away from particular programming languages and program proof systems. We can apply these solvers so long as we have a VC generator that produces VC's as (generalized) Horn formulas. Moreover, as we will see, they give us considerable flexibility to generate VC's in a way that guides or constrains the solution.

In this paper, we will consider such techniques, and show that by writing the VC's in an appropriate form, we can convey domain knowledge about the problem in a way that can simulate the effect of various abstract domains, without having to design and implement a custom program analysis, and without having to specify many of the parameters of these domains, as they can be inferred automatically by the solver. This makes it possible to experiment rapidly with various analyses. Based on this experience, one could then implement a custom analysis (say, using parameters of the abstract domain discovered by the solver for a class of programs) or simply apply the Horn solvers directly if their performance is adequate.

In particular, in this paper we will show that the process of solving for program proofs can be guided by choosing the logical form of the unknown assertions. To obtain universally quantified invariants in this way, we extend Horn solvers to handle *quantified predicates*. That is, we allow an unknown predicate P to appear in a VC as $\forall X.P(\boldsymbol{t}[X])$, where $\boldsymbol{t}[X]$ is a vector of terms with free variables X. This allows us to replace the invariant $P(a, i)$ in our example with, say, $\forall j.\ P(j, a[j], i, N)$. This tells us, in effect, that we have to write the invariant as a formula universally quantified over j, using just the terms j, $a[j]$, i and N. This new invariant assertion gives us the VC's shown in Figure 1, with quantified predicates.

One solution for these VC's is:

$$P(j, x, i, N) \equiv 0 \leq j < i \implies x = 0$$

Notice that substituting this definition into $\forall j.\ P(j, a[j], i, N)$ gives us $\forall j.\ 0 \leq j < i \implies a[j] = 0$, exactly the invariant we obtained previously.

The advantage of using this formulation of the problem is that it constrains the space of solutions in a way that causes the solver to converge, whereas with the more general formulation it may not converge. Another way to say this is that we have provided a restricted language for the inductive invariant. This restriction thus takes the heuristic role of an abstract domain, even though the solver itself may not be based on abstract interpretation. Note, though, that it is a rich abstract domain, since it does not restrict the Boolean structure of P or the set of arithmetic terms or relations that may occur in P.

We will show experimentally that, solving for quantified invariants in this form, we can simulate the effect of different abstract domains, such as the array segmentation domain of [6] and the Fluid Updates array abstraction of [9]. That is, we can induce a Horn clause solver to produce inductive proofs in a language at least as rich as these abstract domains, without implementing a custom program analysis, and without providing abstract domain parameters such as the segment boundary terms or data abstractions. Rather, we simply write the invariants to be solved for in an appropriate form. Moreover, quantified predicates provide us flexibility to solve problems that cannot be solved in these domains.

Related work Quite a variety of approaches have been taken to generation of quantified inductive invariants. Here, we will survey some of these methods and compare them to the present one. One line of work is based on interpolants. Quantification is used for interpolants describing unbounded ranges of arrays [23]; and in more recent work [2] by leveraging Model Checking Modulo Theories. Super-position theorem provers work directly with clauses corresponding to quantified formulas and this has been leveraged for extracting quantified invariants [28], [18]. Common to these approaches, and different from the current work, is that they aim to produce quantified invariants directly. The shape of quantification is left mostly unrestricted. In contrast, the current work takes as starting point a template space of quantified invariants and reduces the problem to quantifier-free invariant generation. A number of abstract interpretation methods have domains that represent universally quantified facts [14, 11, 5, 31]. Here, we aim to avoid the explicit construction of abstract post operators, widenings and refinement procedures needed in these approaches. The constraint-based approach of [25] synthesizes a class of universally quantified linear invariants of array programs. We synthesize a broader class of invariants, without hints as to the index terms occurring in the invariants. In principle, however, a constraint-based linear invariant generator can be a back-end solver in our procedure.

There also exist solving techniques that support existential quantification in Horn clauses, i.e., a quantifier alternation in the form of forall/exists [3]. This approach computes witnesses to existential quantification in form of Skolem relations. The method also relies on a back-end Horn clause solver, however, an abstraction refinement loop is used to iteratively discover required witnesses.

That is, in contrast to the method presented here, [3] calls the back-end solver repeatedly.

2 Preliminaries

We use standard first-order logic over a signature Σ of function and predicate symbols of defined arities. Generally, we use a, b, c for constants (nullary function symbols), f, g for functions, P, Q, R for predicates, x, y, z for individual variables, t, u for terms. When we say a formula is *valid* or *satisfiable*, this is relative to a fixed background theory $\mathcal{T}$. A subset of the signature $\Sigma_I \subseteq \Sigma$ is considered to be *interpreted* by the theory. The *vocabulary* of a formula ϕ (the subset of Σ occurring in it) is denoted $L(\phi)$. The variables occurring free in ϕ are denoted $V(\phi)$. We will write $\phi[X]$ and $t[X]$ for a formula or term with free variables X. If P is a predicate symbol, we will say that a P-formula is a formula of the form $P(t_1, \ldots, t_n)$. If R is a set of symbols, we say ϕ is R-free when $L(\phi) \cap R = \emptyset$.

3 The quantified relational post-fixed point problem

We now give a precise definition of the problem to be solved.

Definition 1. *A* constrained Horn clause *(in the sequel* CHC*) over a vocabulary of predicate symbols $\mathcal{R}$ is a formula of the form $\forall X.B[X] \Rightarrow H[X]$ where*

- *The* head *$H[X]$ is either a P-formula, or is $\mathcal{R}$-free, and*
- *The* body *$B[X]$ is a formula of the form $\exists Y.\ \phi \wedge \psi_1 \wedge \cdots \psi_k$ where ϕ is $\mathcal{R}$-free and ψ_i is a P-formula for some $P \in \mathcal{R}$.*

The clause is called a query *if the head is $\mathcal{R}$-free, else a* rule*. A rule with body* TRUE *is a* fact*.*

Definition 2. *A* relational post-fixed point problem *(RPFP) is a pair $(\mathcal{R}, \mathcal{C})$, where $\mathcal{R}$ is a set of predicate symbols (called the* nodes*) and $\mathcal{C}$ is a set of CHC's over $\mathcal{R}$.*

An RPFP is satisfiable if there is an interpretation of the predicate symbols $\mathcal{R}$, such that each constraint in $\mathcal{C}$ is true under the interpretation. Thus, an interpretation provides a *solution* to an RPFP. We are here interested in effective ways to establish satisfiability of RPFPs and will search for interpretations that can be expressed as formulas using the existing vocabulary. We call the resulting solutions *symbolic solutions*, explained next.

We will refer to $\Sigma \setminus \mathcal{R}$ as the *background vocabulary*. A *symbolic relation* is a term of the form $\lambda.\bar{x}.\ \phi[\bar{x}]$ where $\bar{x}$ is a vector of distinct variables, such that $L(\phi) \subseteq \Sigma \setminus \mathcal{R}$ (that is, ϕ is over only the background vocabulary). A symbolic relational interpretation σ over $\mathcal{R}$ is a map from symbols in $\mathcal{R}$ to symbolic relations of the appropriate arity. If ψ is a formula, we write $\psi\sigma$ to denote ψ with $\sigma(R)$ substituted for each $R \in \mathcal{R}$ and β-reduction applied. For example, if ψ is $R(a, b)$ and $\sigma(R)$ is $\lambda x, y.\ x < y$, then $\psi\sigma$ is $a < b$.

Definition 3. *A* symbolic solution *of RPFP* $(\mathcal{R}, \mathcal{C})$ *is a symbolic relational interpretation over* $\mathcal{R}$ *such that, for all* $C \in \mathcal{C}$, $C\sigma$ *is valid (relative to theory* $\mathcal{T}$*).*

A subtle point worth noting here is that a solution of an RPFP depends on the interpretation of the background symbols. If the background theory is complete (meaning it has a unique model up to isomorphism) then this gives a unique interpretation of $\mathcal{R}$. We can therefore think of an RPFP as a special case of an SMT problem. If $\mathcal{T}$ is incomplete, however (say it includes uninterpreted function symbols) then the symbolic solution effectively gives an interpretation of $\mathcal{R}$ for each theory model. This allows us to leave aspects of the program semantics (say, the heap model) incompletely defined, yet still prove the program correct.

3.1 Refutations and derivation trees

If a solution of the VC's corresponds to a proof of the program, one might ask what corresponds to a counterexample (that is, a proof the program is incorrect). One way to view this is that a set of rules has a minimal model, that is, a least interpretation of the predicate symbols by set containment that satisfies the rules. An RPFP is satisfiable exactly when the minimal model of its rules satisfies all of its queries. The minimal model of the rules is the set of ground facts that can be derived from ground instances of the rules by unit resolution. An RPFP can thus be refuted (proved unsatisfiable) by a ground derivation of False.

As an example, consider the following RPFP:

$$x = y \implies P(x, y) \tag{7}$$

$$P(x, y) \wedge z = y + 1 \implies P(x, z) \tag{8}$$

$$P(x, y) \wedge P(y, z) \implies Q(x, z) \tag{9}$$

$$Q(x, z) \implies x + 2 \leq z \tag{10}$$

We can think of these formulas as the VC's of a program with two procedures, P and Q. Procedure P is recursive and either returns its input (7) or calls itself and returns the result plus one (8). Procedure Q calls P twice (9). The query represents a specification that Q increments its argument by at least two (10). The program does not satisfy this specification, which we can prove by the following ground derivation. First, from (7), setting $x = y = 0$, we derive the ground fact $P(0, 0)$. Then from (9), setting $x = y = z = 0$, we obtain $P(0, 0) \wedge P(0, 0) \implies Q(0, 0)$. Resolving with $P(0, 0)$, we derive $Q(0, 0)$. Then from (10), setting $x = z = 0$, we obtain $Q(0, 0) \implies$ False. Resolving with $Q(0, 0)$, we obtain False. This refutation can also be thought of as an execution of the program that does not satisfy the specification.

We can discover a refutation by constructing a *derivation tree*, by a process that is essentially logic program execution. A derivation tree is obtained by starting with a query and successively unifying $\mathcal{R}$-predicates in bodies with heads of rules until we reach facts. In our example, we might obtain the derivation tree

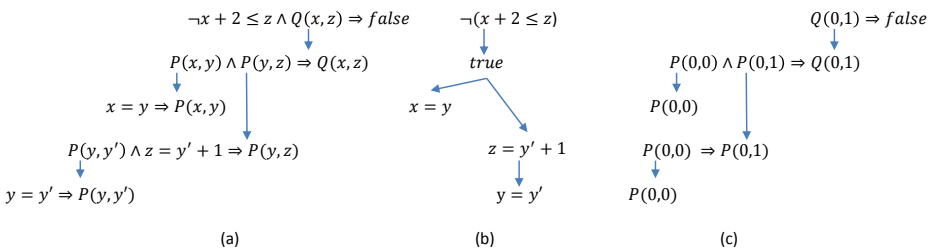

Fig. 2. Deriving a refutation (a) Derivation tree. Arrows represent unification steps. (b) Constraint tree. (c) Resulting ground refutation.

of Figure 2(a). Extracting the constraints from these clauses, we obtain the constraint tree shown in Figure 2(b). If these constraints are satisfiable, substituting the satisfying assignment into the derivation tree gives us a ground derivation of FALSE, as shown in Figure 2(c). On the other hand, showing satisfiability of the RPFP is equivalent to proving unsatisfiability of all possible derivation trees. This view will be useful later when we discuss quantifier instantiation.

3.2 Solving RPFP's

A variety of techniques can be applied to solve RPFP's symbolically or produce refutations. For example, ARMC [13] applies predicate abstraction [12]. Given a set $\mathcal{P}$ of atomic predicates, it synthesizes the strongest map from $\mathcal{R}$ to Boolean combinations (alternatively cubes) over $\mathcal{P}$ that solves the problem. The predicates are derived from interpolants for unsatisfiable derivation trees. Various other methods are available [30, 17] but all synthesize the solution in some manner from solutions of finite unwindings of the clause set.

All these methods may diverge by producing an infinite sequence of approximations to a solution. For example, in our array initialization example above, we may first consider just one iteration of the loop, deriving an approximation of the loop invariant $P(i, a, N)$ that involves the predicate $a[0] = 0$. Then we consider two iterations of the loop, obtaining $a[1] = 0$ and so on *ad infinitum*. In short, we need some way to tell the tool that the invariant must involve a quantifier.

3.3 Quantified predicates

To do this, we will allow our symbolic program assertions to contain quantifiers. We say that a $\forall P$-formula is a formula of the form $\forall Y.P(\boldsymbol{t})$. We extend constrained Horn clauses to *quantified* Horn clauses as follows:

Definition 4. *A* quantified Horn clause *(in the sequel* QHC*) over a vocabulary of predicate symbols $\mathcal{R}$ is a formula of the form $\forall X.B[X] \Rightarrow H[X]$ where*

- *The* head $H[X]$ *is either a $\forall P$-formula, or is $\mathcal{R}$-free, and*

– *The* body $B[X]$ *is a formula of the form* $\exists Y.\ \phi \wedge \psi_1 \wedge \cdots \psi_k$ *where* ϕ *is* $\mathcal{R}$*-free and* ψ_i *is a* $\forall P$*-formula for some* $P \in \mathcal{R}$*.*

The only difference from the previous definition is that we use $\forall P$-formulas in place of P-formulas. This allows us to express VC's such as those in our second version of the array initialization problem (4–6).

We first observe that a universal quantifier in the head of a rule poses no difficulty, as it can simply be shifted to prenex position. That is, the formula

$$\forall X.\ B[X] \implies \forall Y.P(\boldsymbol{t})$$

is equi-valid to

$$\forall X, S.\ B[X] \implies P(\boldsymbol{t})\langle S/Y\rangle$$

where S is a set of fresh variables. The difficulty lies, rather, in the $\forall P$-formulas that occur as conjuncts in the body $B[X]$. We can think of these formulas as representing an infinity of groundings.

As a result, derivation trees are now infinitely branching. Consider, for example, this rule:

$$(\forall y.\ P(x, y)) \implies Q(x)$$

The rule requires an infinite set of tuples $P(x, y)$ to derive a single tuple $Q(x)$. One can also easily construct cases where the only feasible derivation tree has infinite height. For example, in the theory of arithmetic:

$$\begin{aligned} x = 0 &\implies P(x) \\ P(x) &\implies P(x+1) \\ (\forall 0 \le y.\ P(y)) &\implies Q \end{aligned}$$

A derivation of Q requires an infinite set of subtrees corresponding to $P(0), P(1), P(2), \ldots$, where the derivation of $P(k)$ is of height $k+1$. Thus, the height of the derivation of Q is ω.

A similar example illustrates that finite quantifier instantiation is incomplete for establishing satisfiability of RPFP's. Consider the system below.

$$\begin{aligned} &P(0,1) \\ P(x,y) &\implies P(x, y+1) \\ P(x,y) &\implies P(x+1, y+1) \\ (\forall x\ .\ P(x,y)) &\implies Q(y) \\ Q(x) &\implies \textsc{False} \end{aligned} \tag{11}$$

It exploits that compactness does not hold for the theory of natural numbers. It is satisfiable with symbolic solution $P(x, y) \equiv x < y$, $Q(y) \equiv \textsc{False}$; yet every finite instantiation $P(a_1, y) \wedge \ldots P(a_n, y) \implies Q(y)$ of the quantified

clause (11) produces a stronger system that is unsatisfiable. We have yet to encounter applications from program analysis where this source of incompleteness is significant.

The approach we will take to quantifiers is the same as is typically taken in SMT solvers: we will replace a universally quantified formula by a finite set of its instances. Say that an *instantiation* of a formula $\forall Y.\ P(\boldsymbol{t})$ is $P(\boldsymbol{t})\langle S/Y\rangle$ for some vector of terms S. We can show:

Theorem 1. *Let Π be a quantified RPFP and let Π' be an unquantified RPFP obtained by shifting quantifiers in the heads of the rules to prenex position and replacing each $\forall P$ formula in a body by a finite conjunction of its instantiations. Then a solution of Π' is also a solution of Π.*

Proof A conjunction of instantiations of a universal formula ϕ is implied by ϕ . Thus, the body of a QHC in the instantiated Π' is implied by the corresponding body in Π. Since the bodies are on the left-hand side of the implications, it follows that the constraints in Π' imply the corresponding constraints in Π. A solution of Π' is thus a solution of Π. □.

This means that if we replace any $\forall P$-formula with a finite instantiation, any proof of the program we obtain is valid. However, a counterexample may not be valid. By adding one more instance, we may find the counterexample ruled out. Thus, finite instantiation is a conservative abstraction.

In this paper we will consider a simple syntactic approach to generate finite instantiations, relying on Theorem 1. It uses pattern-matching heuristics to instantiate the universals in the QHC bodies. The resulting problem Π' can then be solved by any of the available RPFP solvers. Any solution of Π' is also a solution of Π and thus implies correctness of the program. This approach has the advantage that it leaves the solver itself unchanged. Thus, we can apply any available solver for unquantified Horn clauses.

4 Trigger-based quantifier instantiation

The syntactic approach can be applied as a pre-processing step on the RPFP. It operates in a matter that is inspired by quantifier instantiation in SMT solvers [8]. We begin with the theory of equality and uninterpreted functions, and then extend to cover the theory of arrays. For the programs we have analyzed here, it was sufficient to treat arithmetical symbols $+, \times$ as uninterpreted.

4.1 Theory of equality

Trigger-based instantiation [8] is a method for instantiating universal quantifiers. The *current goal* of the prover is a set of literals whose satisfiability is to be determined. Quantifiers in the current goal are annotated by triggers, that is, terms containing all the quantified variables. A *match* is any variable substitution that takes a trigger to an existing ground term from the current goal, modulo the set of asserted equalities in the current goal.

$$i_0 = 0 \implies P(a_0[k_0], i_0) \tag{12}$$
$$\forall j_1.\ P(a_1[j_1], i_1) \wedge i_1 < N_1 \implies P(a_1[i_1 \leftarrow 0][k_1], i_1 + 1) \tag{13}$$
$$\forall j_2.\ P(a_2[j_2], i_2) \wedge \neg(i_2 < N_2) \implies 0 \leq k_2 < N_2 \implies a_2[k_2] = 0 \tag{14}$$

Fig. 3. VC's generated for array initialization, after shifting head quantifiers to prenex.

The intuition behind trigger-based instantiation as stated in [8] is that an instance of a universal is likely to be relevant if it contains many terms that can be proved equal to existing terms. Here, unlike in [8], we wish to find such instances in advance of doing any actual deduction. Thus we will produce an *over-approximation* of the equality relation, considering terms equal if they *may* be proved equal in some propositional case. Moreover, we wish to produce enough instantiations to make all of the derivation trees unsatisfiable. Thus the proofs we are considering are proofs of unsatisfiability of the constraints in these trees. For this reason, we will consider two terms possibly equal if two instances of these terms may be inferred to be equal in some derivation tree.

To discover instantiations of $\forall P$-formulas in quantified clauses we build a structure called an E-graph [8]. The E-graph is a term DAG with an equivalence relation over the nodes. It provides a compact representation of a collection of terms that might be generated by instantiating axioms of the theory of equality. We produce instantiations by matching terms containing bound variables against terms represented in the E-graph.

Consider, for example, our array initialization problem of Figure 1. For now, we consider the array operators $\cdot[\cdot]$ and $\cdot[\cdot \leftarrow \cdot]$ to be uninterpreted functions. We begin by shifting head quantifiers to prenex (introducing fresh variables k_i) to obtain the QHC's of Figure 3. We also rename the bound variables so they are distinct.

The E-graph we obtain from these formulas is shown in Figure 4. It contains all of the terms occurring outside of $\forall P$-formulas. It merges the terms i_0, 0 and $a_2[k_2]$ into an equality class, due to the presence of equalities in the formulas. Then we try to match certain terms within $\forall P$-formulas called *triggers* with terms represented in the E-graph, to produce instantiations of the variables. We never match bare variables, hence the matched terms must always agree on at least the top-level function symbol. In practice, we use array select terms as triggers. The process of matching triggers against the E-graph to produce instantiations is called E-matching. The exact set of matching terms we produce from the E-graph is a heuristic choice.

In the example, we can match $a_2[j_2]$ with the E-graph term $a_2[k_2]$, using the unifier $j_2 = k_2$. We use this assignment to instantiate the quantifier in (14). This gives us the instance $P(a_2[k_2], i_2)$. We now merge the arguments of P in this term with corresponding arguments in the heads of the rules. Thus, we merge $a_2[k_2]$ with $a_1[i_1 \leftarrow 0][k_1]$ and with $a_0[k_0]$, and we merge i_2 with $i_1 + 1$ and i_0 in the E-graph. We do this because these terms might be unified in the construction

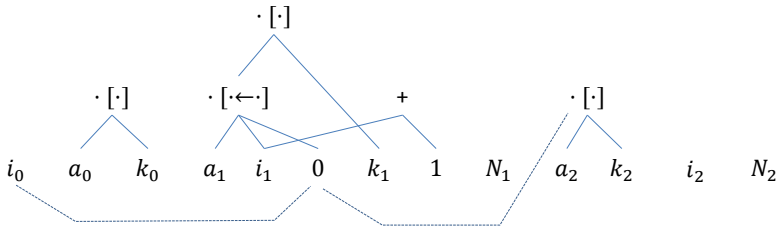

Fig. 4. Starting E-graph for formulas of Figure 3. Dashed lines represent equality classes.

of a derivation tree. In principle this merge might give us new matches, but in this case no new matches result. In particular, there is no match for $a_1[\cdot]$, so we still have no instances of P in the body of rule (13), leaving the instantiated system unsolvable.

4.2 Theory of arrays

To solve this problem, we need to attach semantics to the array operations select and store. In particular, we need to take into account that $a[x \leftarrow y][z]$ may equal $a[z]$ (in the case when $x \neq z$). We do this by adding the axioms of the array theory to the mix. These are:

$$\forall a, i, d.\ a[i \leftarrow d][i] = d$$
$$\forall a, i, j, d.\ (i = j) \vee a[i \leftarrow d][j] = a[j]$$

The trigger associated with the first axiom is $a[i \leftarrow d][i]$. The second axiom contains the sub-term $a[i \leftarrow d][j]$ that contains all the bound variables and we use it as a trigger. In our example, this trigger matches the term $a_1[i_1 \leftarrow 0][k_1]$ in the E-graph, producing the instance $(i_1 = k_1) \vee a_1[i_1 \leftarrow 0][k_1] = a_1[k_1]$. This contributes the term $a_1[k_1]$ to the E-graph, which then matches $a_1[j_0]$ to give the instance $P(a_1[k_1], i_1)$. Adding this instance, we merge $a_1[k_1]$ with $a_1[i_1 \leftarrow 0][a_0]$ and i_1 with $i_1 + 1$ in the E-graph. This results in no further matches, so instantiation terminates. Note that matching with the first axiom never produces new terms, so we need not consider it. The final E-graph after quantifier instantiation is shown in Figure 5.

Now, replacing the $\forall P$-formulas with the obtained instantiations, we have the instantiated problem Π' shown in Figure 6. This problem has a solution, for example, the same solution we obtained for the original quantified problem:

$$P(j, x, i, N) \equiv 0 \leq j < i \implies x = 0$$

One can verify this solution by plugging in the definition of P and checking validity of the resulting quantifier free formulas with an SMT solver.

As a side-remark we note that trigger-based quantifier instantiation becomes a complete decision procedure for the existential theory of non-extensional arrays

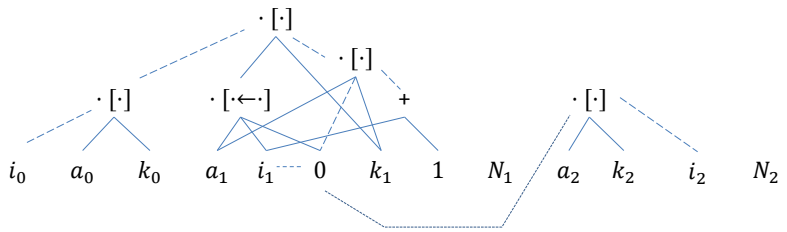

Fig. 5. Final E-graph for formulas of Figure 1.

$$i_0 = 0 \implies P(a_0[k_0], i_0) \tag{15}$$
$$P(a_1[k_1], i_1) \wedge i_1 < N_1 \implies P(a_1[i_1 \leftarrow 0][k_1], i_1 + 1) \tag{16}$$
$$P(a_2[k_2], i_2) \wedge \neg(i_2 < N_2) \implies 0 \leq k_2 < N_2 \implies a_2[k_2] = 0 \tag{17}$$

Fig. 6. VC's generated for array initialization example, after instantiation.

if we also add the two terms $a[i \leftarrow d], a[j]$ as a joint trigger (called multi-pattern) to the second array axiom. But it is well recognized that this axiom is irrelevant for verification conditions from programs (where the same array does not get updated in two different ways), so we disregard this trigger in our experiments.

4.3 Algorithm

Figure 7 shows a high-level algorithm for trigger-based instantiation, based on standard techniques for E-matching [8]. We start by shifting quantifiers, then collect the $\forall P$-formulas from the bodies, adding axioms of the theory. These are the universal formulas to be instantiated. Then we build an E-graph, merging terms based on equalities in the formulas. We then enter a loop, instantiating the quantified formulas using the E-graph. Each time an instance matches the head of a rule, we merge the corresponding arguments in the E-graph. Finally, for each $\forall P$-formula in a rule C, we gather the instances that are expressed using the variables of C (thus if the instance contains a variable from another constraint, we reject it as irrelevant). The conjunction of these instances replaces the $\forall P$-formula.

In principle, we may have a matching loop. For example, suppose we have the quantified formula $\forall y.\ P(g(y), g(f(y)))$ in $\mathcal{P}$ and associated trigger $g(y)$ and a term $g(a)$ in the E-graph. We can then obtain an infinite sequence of instantiations: First y is instantiated with a producing the grounding $P(g(a), g(f(a)))$. The E-graph is updated with the term $g(f(a))$ terms $g(a), g(f(a)), g(f(f(a))), \ldots$ Though we have not seen this in practice, it would be possible to terminate such loops by putting a bound on term size.

As in SMT solvers, the trigger-based instantiation approach is heuristic. It may or may not generate the instances needed to obtain a proof. Moreover, we

Algorithm *Trigger-instantiate*
Input: a set of QHC's $\mathcal{C}$
Output: instantiations of $\mathcal{C}$

Shift head quantifiers in $\mathcal{C}$ to prenex position.
Let $\mathcal{P}$ be the set of $\forall P$-formulas in $\mathcal{C}$.
Let $\mathcal{Q}$ be $\mathcal{P}$ with the array theory axioms added.
Let E be an empty E-graph.
Add the ground terms of $\mathcal{C}$ to E.
For every ground equality $x = y$ in $\mathcal{C}$, merge x and y in E.
Let $\mathcal{I}$ be the empty set of formulas.
Repeat:
    Let $\mathcal{G}$ be the instances of $\mathcal{Q}$ obtained by E-matching with E.
    Add $\mathcal{G}$ to $\mathcal{I}$.
    Add the ground terms of $\mathcal{G}$ to E.
    For every instance $G \in \mathcal{G}$ of some formula $\forall Y.\ P(\boldsymbol{t})$ in $\mathcal{P}$, do
        For every head $P(\boldsymbol{u})$ of a rule in $\mathcal{C}$, do
            Merge $\boldsymbol{t}$ with $\boldsymbol{u}$ in E. (*)
Until no change in $\mathcal{I}$.
For each formula ϕ in $\mathcal{P}$, where ϕ occurs in QHC $C \in \mathcal{C}$ do
    Let $\mathcal{I}_\phi$ be the set of instances ψ of ϕ in $\mathcal{I}$ s.t. $V(\psi) \subseteq V(C)$
        Substitute $\wedge\mathcal{I}_\phi$ for ϕ in C
Return $\mathcal{C}$.

Fig. 7. Trigger-based instantiation algorithm

must view the counterexamples (*i.e.*, derivation trees) generated by the solver as suspect, since they contain only a finite subset of the instances of the universals. We will observe in Section 5, however, that a reasonably precise analysis can be obtained with these fairly simple heuristics.

In fact, in our experiments, we have found a more restrictive instantiation policy to be adequate. That is, we only instantiate variables with *existing* terms (not all terms represented in the E-graph) and we instantiate each QHC in isolation. Reducing the number of irrelevant instances in this way tends to improve the solver performance.

In summary we have the following variants and restrictions of Algorithm *Trigger-instantiate*:

Inter vs. intraprocedural: The algorithm takes a set $\mathcal{C}$ of Horn clauses as input. In one extreme take $\mathcal{C}$ as the entire set of clauses to be checked for satisfiability. In the other extreme process each clause C_i by setting $\mathcal{C}$ to $\{C_i\}$ and invoke the instantiation algorithm.

Instantiation vocabulary: The E-graph represents potentially an infinite number of terms. For example, if a and $f(a)$ are merged in E, then E encodes that $f(f(a)) = a$. Each of the terms $f(a)$, $f(f(a))$, $f(f(f(a)))$, etc. can be used as representatives for a. Different instances of Algorithm *Trigger-instantiate* are obtained by bounding the number of representatives admitted for each

match. Since the algorithm uses *may* equality, it becomes highly incomplete if only a single match is admitted.

Instantiation depth: SMT solvers manage quantifier instantiations using a priority queue of instances. An instantiation is *heavy* if it uses terms that were constructed by a long sequence of instantiations. Lighter (younger) terms are preferred. Likewise, our pre-processing instantiation can cut off instantiations based on a maximal weight.

5 Applications

We now consider some applications of quantified RPFP solving. We will observe that by choosing our symbolic invariants appropriately, we can simulate the effect of some abstract domains from the literature. In this way, we can avoid implementing a custom analysis. As we sill see, we can also avoid the problem of tuning the parameters of the abstract domain, since the values of these parameters can be extracted automatically.

5.1 Simulating array segmentation abstractions

Cousot, Cousot and Logozzo describe an *array segmentation functor* for analyzing programs that manipulate arrays [6]. This is a parametrized class of abstract domains that characterize the contents of an array. An abstract array is divided into segments, that is, consecutive subsets of the array elements that are divided by symbolic index expressions. The elements in each segment are characterized by a chosen abstract domain on data. Both the segment boundary expressions and the data domain may depend on scalar variables in the program.

As an example, the following expression represents an array that contains a segment of zeros for indices $0 \ldots i-1$ and a segment of arbitrary values for indices $\geq i$:

```
{0} 0 {i}? T {A.Length}?
```

The symbol `T` stands for the top element of the data domain. The question marks indicate that the preceding segment may be empty. This abstract array can be expressed in logical form as

$$\forall x.\ 0 \leq \mathtt{i} \leq \mathtt{A.Length} \wedge (0 \leq x < i \implies \mathtt{A}[x] = 0)$$

In fact, *any* abstract array can be expressed by a predicate in integer linear arithmetic of the form $\forall x.\ P(x, A[x], \boldsymbol{v})$ where A is the array and $\boldsymbol{v}$ are the scalar variables of the program, provided the data domain is expressible in the logic. Now suppose we decorate a program with symbolic invariants of this form, for every array A in the program. It follows that if the array segmentation functor can prove the program for some value of its parameters, then the resulting RPFP has a solution, thus we can in principle prove the program using an RPFP solver.

```
public Random(int Seed) {
  Contract.Requires(Seed != Int32.MinValue);
  int num2 = 161803398 - Math.Abs(Seed);
  this.SeedArray = new int[56];
  this.SeedArray[55] = num2;
  int num3 = 1;
  // Loop 1
  for (int i = 1; i < 55; i++) {
    int index = (21 * i) % 55;
    this.SeedArray[index] = num3; // (*)
    num3 = num2 - num3;
    if (num3 < 0) num3 += 2147483647;
    num2 = this.SeedArray[index];
  }
  Contract.Assert(Contract.Forall( // (**)
     0,this.SeedArray.Length - 1, i => a[i] >= -1));
  // Loop 2
  for (int j = 1; j < 5; j++) {
    // Loop 3
    for (int k = 1; k < 56; k++) {
      this.SeedArray[k] -= this.SeedArray[1 + (k + 30) % 55];
      if (this.SeedArray[k] < 0)
        this.SeedArray[k] += 2147483647;
  } }
  Contract.Assert(Contract.Forall(0, // (***)
     this.SeedArray.Length, i => a[i] >= -1));
}
```

Fig. 8. Motivating example for Array Segmentation Functor

As a motivating example, consider the program from [6], shown in Figure 8. When the array `this.SeedArray` is created, it is implicitly initialized to all zero. After the first loop, we must prove that all the elements of array but the last are ≥ -1 (this is the semantics of `Contract.Forall`). We decorate Loop 1 with a symbolic invariant $\forall x.\ P(x, \texttt{this.SeedArray}[x], \boldsymbol{v})$, where $\boldsymbol{v}$ contains the program's integer variables. We then generate the verification conditions for the loop. The VC's are encoded into integer arithmetic. Since the actual program variables are bit vectors, we must test each integer operation for overflow. This gives us an RPFP Π with one unknown predicate P. We instantiate Π using trigger-based instantiation to get the quantifier-free problem Π'. We then used Duality to solve Π' for P, obtaining the following inductive invariant for the

loop:

$$\forall x.\ \begin{pmatrix} (\texttt{num2} \geq -2147483648 \vee \texttt{num2} \leq 2147483647) \\ \wedge (0 \leq \texttt{num3} + 1) \\ \wedge (0 \leq \texttt{this.SeedArray}[x] + 1 \vee 0 \leq x - 55) \end{pmatrix}$$

The first conjunct of this invariant is equivalent to TRUE. The second says that the scalar `num3` is ≥ -1 while the third says that the array value is ≥ -1 for all all indices < 55. Note, this applies to negative indices as well, though it would be a run-time error to access these.

Notice that we obtained this result using a generic solver, without implementing a custom analysis. Notice also that the segment boundary 55 and the data predicate $0 \leq \texttt{this.SeedArray}[x] + 1$ were generated in the solving process, so we did not have to provide these as a parameter of the abstraction. The runtime of the tool was 0.3 seconds. This is undoubtedly slower than the custom analysis of [6]. On the other hand, it might be fast enough for some applications. The method can similarly solve for an inductive invariant of the second loop.

A more significant advantage to this approach is the flexibility it provides to experiment with abstractions. For example, suppose we need an invariant that relates the values of corresponding elements of two arrays a and b. We could use the symbolic invariant $\forall x.\ P(x, a[x], b[x], \boldsymbol{v})$. Or suppose we need to relate distant elements of the arrays. We could then use two quantifiers, expressing the desired invariant in the form $\forall x, y,\ (x, a[x], y, b[y], \boldsymbol{v})$. This would allow us to express the fact, for example, that a is the reverse of b, as follows: $\forall x, y.\ 0 \leq x < N \wedge y = N - x \implies a[x] = b[y]$. To implement this using the array segmentation functor would be a significant manual effort, as we would have to implement a component abstract domain that names array segments with a corresponding segment unification procedure, introduce auxiliary variables to represent array segment values and introduce the appropriate terms and relations, including the relation $y = N - x$, into the scalar abstraction. Here, we simply change the form of the symbolic invariant that annotates the program. The necessary terms and predicates can be synthesized by the Horn solvers.

5.2 Simulating the Fluid Updates abstraction

The Fluid Updates method of Aiken, Dillig and Dillig [9] provides a richer abstract domain for arrays. The abstraction can symbolically represent pair-wise points-to relations between arrays. A points-to relation is represented by a triple $a[x] \to \phi \to b[y]$, where a and b are symbolic terms representing locations and containing parameters x and y, and ϕ is a may- or must-condition for $a[x]$ to point to $b[y]$. For example, to write that all the elements of the array pointed to by a up to $i - 1$ are zero, we would write a must relation $(*a)[j] \to 0 \leq j < N \to *0$, where we think of the constant zero as a pointer to a zero object.

We can express all such relationships logically using predicates of the form $\forall x, y.\ P(x, a[x], b[y], \boldsymbol{v})$, where $a[x]$ and $b[y]$ are location terms and $\boldsymbol{v}$ contains

scalar program variables. For example, the must relationship

$$a[x] \to 0 \leq x < N \wedge y = N - x \to b[y]$$

says that element x of array a must point to element $N - x$ of array b. This can be expressed as

$$\forall x, y.\ 0 \leq x < N \wedge y = N - x \implies \text{pto}(a[x], b[y])$$

where pto represents the points-to relation. The corresponding may relationship is

$$\forall x, y.\ \text{pto}(a[x], b[y]) \implies 0 \leq x < N \wedge y = N - x.$$

In the case where $b[x]$ is $*t$ for some integer-valued term t, the relation pto becomes simply equality over integers.

Using this scheme, we can capture the information available in the Fluid Updates abstraction by simply decorating the program with symbolic invariants of the form $\forall x, y.\ P(x, a[x], b[y], \boldsymbol{v})$, for any pairs of location terms $a[x]$ and $b[y]$ we wish to track. We can, if desired, narrow down the pairs tracked using any available points-to analysis.

To test this idea, we constructed the VC's manually for the set of synthetic test programs used in [9]. We used the quantifier instantiation procedure of section 4 in the intraprocedural mode, with the restrictive instantiation policy. No matching loops were observed. Table 1 shows the performance we obtained solving these instantiated VC's using the Horn solving engine of Z3 [17], ARMC [13], and Duality, compared to the results of [9]. Run times should be considered approximate as machine speeds may differ slightly. A question mark indicates the tool could not process the problem. In Duality, we used a recent interpolation technique that handles linear rational arithmetic [1]. It is slower, but has better convergence behavior than the proof-based method of [29]. Integer arithmetic and the theory of arrays are handled by eager axiom instantiation. We observe that each tool is able to solve most of the problems, though the performance is not always as good as the Fluid Updates method. All tools fail for one case (`init_even`). This requires a divisibility constraint in the invariant, which none of the tools supports. All the other problems can be solved by at least one tool. Thus, using generic Horn solvers, we obtain results similar to what can be obtained using a specialized abstract domain.

Again, however, we observe that using symbolic invariants gives us flexibility not available in an analysis tool implementing a particular abstract domain. Consider, for example, the following simple fragment:

var i : int := 0;
while $i < N$ **do**
    $c[i]$:= a[i] - b[i];
    $i := i + 1$;
done
assert $\forall 0 \leq j < N.\ c[j] = a[j] - b[j]$;

example	[9]	Z3 Horn	ARMC	Duality
init	0.01	0.06	0.15	0.72
init_non_constant	0.02	0.08	0.48	6.60
init_partial	0.01	0.03	0.14	2.60
init_partial_BUG	0.02	0.01	0.07	0.03
init_even	0.04	TO	?	TO
_2Darray_init	0.04	0.18	?	TO
copy	0.01	0.04	0.20	1.40
copy_partial	0.01	0.04	0.21	1.80
copy_odd	0.04	TO	?	4.50
reverse	0.03	0.12	2.28	8.50
reverse_BUG	0.04	0.01	0.08	0.03
check_swap	0.12	0.41	3.0	40.60
check_double_swap	0.16	1.37	4.4	TO
check_strcpy	0.07	0.05	0.15	0.62
check_strlen	0.02	0.07	0.02	0.20
check_strlen_BUG	0.01	0.07	?	0.03
check_memcpy	0.04	0.04	0.20	16.30
find	0.02	0.01	0.08	0.38
find_first_nonnull	0.02	0.01	0.08	0.39
array_append	0.02	0.04	1.76	1.50
merge_interleave	0.09	0.04	?	1.50
alloc_fixed_size	0.02	0.02	0.09	0.69
alloc_nonfixed_size	0.03	0.03	0.13	0.42

Table 1. Performance on synthetic array benchmarks. Run times in seconds.

To prove the assertion, we must track the values in three arrays. To extend Fluid Updates to handle this case would require modifying the tool. To handle this example using symbolic invariants, we simply decorate the loop with the predicate $\forall x.\ P(x, a[x], b[x], c[x], i, N)$ and solve for P. If we need to relate distinct indices in the arrays, we can simply add another quantifier.

5.3 Proving termination

Using the generic Horn solver at the back-end allows one to prove termination of array manipulating problem without constructing a specialized termination checker for programs over arrays.

For example, our approach proves termination of the following program.

```
void main () {
  int i, n, x, a[n];
  for(i = 0; i < n; i++) {
    a[i] = 1;
  }
  x = read_int();
  while (x > 0) {
```

```
    for(i = 0; i < n; i++) x = x-a[i];
  }
}
```

Here, termination of the while loop depends on the values stored in the array. We trigger termination proving by requiring that the restriction of the transition relation of the program with the quantified invariants, which needs to be inferred accordingly, is disjunctively well-founded. After the quantifier instantiation step, ARMC [13] proves termination of all loops in 1.7 sec.

6 Conclusion

Program proving can be reduced to solving a symbolic relation post-fixed point problem. We decorate the program with suitable symbolic invariants (which may be loop invariants, procedure summaries, and so on) yielding the verification conditions which can then be *solved* to produce a program proof. Generic solvers exist to solve these problems modulo various background theories.

We observed that by adjusting the form of the desired proof, we can guide the solver to produce a proof when it would otherwise fail. In particular, we extended RPFP solvers to handle *quantified* symbolic invariants. This allows us to solve for invariants within a restricted domain by choosing the form of the invariant, chiefly its quantifier structure. This allows us to simulate existing abstract domains using a generic solver, without having to directly implement these domains or make certain heuristic choices, such as the terms and predicates that form the parameters of the abstract domain. Moreover the approach gives the flexibility to go beyond these domains and experiment quickly with various invariant forms. It also allows us to guide the proof search by using alternative proof decompositions, without changing the underlying solver.

One view of this approach is as a way of rapidly prototyping program analyses. If the performance of the prototype is adequate, we may simply apply the generic logical solver. If not, we may use the results as a guide to a more efficient custom implementation.

We observed that the primary difficulty in solving for quantified symbolic invariants is in *quantifier instantiation*. We apply simple trigger-based heuristics, similar to those used in SMT solvers, but adapted to our purpose. These heuristics were found adequate in some cases, but clearly more work is needed in this area, perhaps applying model-based quantifier instantiation methods, or judiciously leveraging quantifier elimination methods when they are available, or controlling instantiation using an abstraction refinement methodology. This remains for future work, as does the question of synthesizing quantifier alternations.

Finally, we have here explored only a small part of the space of possible applications of such methods. For example, properties involving the shape of heap data structures can in principle be expressed, for example, using a reachability predicate. It remains to be seen whether relational fixed point solving techniques could be applied to generate invariants in rich domains such as this.

Acknowledgements This research was supported in part by ERC project 308125 VeriSynth.

References

1. A. Albarghouthi and K. L. McMillan. Beautiful interpolants. In *CAV*, 2013.
2. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Lazy abstraction with interpolants for arrays. In N. Bjørner and A. Voronkov, editors, *LPAR*, volume 7180 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2012.
3. T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified Horn clauses. In *CAV*, 2013.
4. N. Bjørner, K. L. McMillan, and A. Rybalchenko. Program verification as Satisfiability Modulo Theories. In *SMT*, 2012.
5. P. Cousot. Verification by abstract interpretation. In *Proc. Int. Symp. on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday*, number 2772 in LNCS, pages 243–268. Springer, 2003.
6. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, 2011.
7. G. Delzanno and A. Podelski. Model Checking in CLP. In *TACAS*, pages 223–239, 1999.
8. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3), 2005.
9. I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, 2010.
10. C. Flanagan. Automatic software model checking using clp. In *ESOP*, pages 189–203, 2003.
11. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.
12. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, 1997.
13. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.
14. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In G. C. Necula and P. Wadler, editors, *POPL*, pages 235–246. ACM, 2008.
15. A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free Horn clauses over LI+UIF. In *APLAS*, 2011.
16. M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program development using abstract interpretation (and the ciao system preprocessor). In R. Cousot, editor, *SAS 2003*, volume 2694 of *LNCS*. Springer, 2003.
17. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.
18. K. Hoder, L. Kovács, and A. Voronkov. Case studies on invariant generation using a saturation theorem prover. In I. Z. Batyrshin and G. Sidorov, editors, *MICAI (1)*, volume 7094 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2011.
19. H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A verification toolkit for numerical transition systems - tool paper. In D. Giannakopoulou and D. Méry, editors, *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 247–251. Springer, 2012.

20. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.
21. J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. Tracer: A symbolic execution tool for verification. In *CAV*, pages 758–766, 2012.
22. J. Jaffar, A. E. Santosa, and R. Voicu. Modeling Systems in CLP. In *ICLP*, pages 412–413, 2005.
23. R. Jhala and K. L. McMillan. Array abstractions from proofs. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 193–206. Springer, 2007.
24. A. Komuravelli, A. Gurfinkel, S. Chaki, and E. Clarke. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *CAV*, 2013.
25. D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. SMT-Based Array Invariant Generation. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2013.
26. K. L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1), 2005.
27. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.
28. K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2008.
29. K. L. McMillan. Interpolants from Z3 proofs. In *FMCAD*, 2011.
30. K. L. McMillan and A. Rybalchenko. Computing relational fixed points using interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, 2013. `http://research.microsoft.com/apps/pubs/?id=180055`.
31. A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, pages 82–97, 2001.
32. P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for horn-clause verification. In *CAV*, 2013.