Verification of Programs by Combining Iterated Specialization with Interpolation

E. De Angelis and F. Fioravanti
DEC, University G. d'Annunzio, Pescara, Italy
{emanuele.deangelis, fioravanti}@unich.it

J. A. Navas
NASA Ames Research Center, Moffett Field, USA
jorge.a.navaslaserna@nasa.gov

M. Proietti
IASI-CNR, Rome, Italy
maurizio.proietti@iasi.cnr.it

We present a verification technique for program safety that combines *Iterated Specialization* and *Interpolating Horn Clause Solving*. Our new method composes together these two techniques in a modular way by exploiting the common Horn Clause representation of the verification problem. The Iterated Specialization verifier transforms an initial set of verification conditions by using unfold/fold equivalence preserving transformation rules. During transformation, program invariants are discovered by applying widening operators. Then the output set of specialized verification conditions is analyzed by an Interpolating Horn Clause solver, hence adding the effect of interpolation to the effect of widening. The specialization and interpolation phases can be iterated, and also combined with other transformations that change the direction of propagation of the constraints (forward from the program preconditions or backward from the error conditions). We have implemented our verification technique by integrating the VeriMAP verifier with the FTCLP Horn Clause solver, based on Iterated Specialization and Interpolation, respectively. Our experimental results show that the integrated verifier improves the precision of each of the individual components run separately.

1 Introduction

Constraint Logic Programming[1] (CLP) [30] is becoming increasingly popular as a logical basis for developing methods and tools for software verification (see, for instance, [8, 16, 24, 32, 39, 37]). Indeed, CLP provides a suitable formalism for expressing *verification conditions* that guarantee the correctness of imperative, functional, or concurrent programs and, moreover, constraints are very useful for encoding properties of data domains such as integers, reals, arrays, and heaps [6, 18, 11, 35]. An advantage of using a CLP representation for verification problems is that we can then combine reasoning techniques and constraint solvers based on the common logical language [22]. In this paper we will show that, by exploiting the CLP representation, we can combine in a modular way a verification technique based on CLP *specialization* [16] and a verification technique based on interpolation [23].

The use of CLP program specialization, possibly integrated with abstract interpretation, has been proposed in various verification techniques [5, 16, 37, 38]. In particular, *iterated specialization* [16] makes use of *unfold/fold* CLP program transformations to specialize a given set of verification conditions with respect to the constraints representing the *initial* and *error* configurations. The objective of the specialization transformation is to derive an equivalent set of verification conditions represented as a finite set of constrained facts from which one can immediately infer program correctness or incorrectness. Since the

[1] In this paper we use interchangeably the terminology "Constraint Logic Program" and "Constrained Horn Clauses" [8].

Submitted to:
HSVC 2014

© E. De Angelis, F. Fioravanti, J. A. Navas & M. Proietti

verification problem is undecidable in general, the specialization strategy employs a suitable *generalization strategy* to guarantee the termination of the analysis. Generalization consists in replacing a clause `H :- c,B` by the new clause `H :- c,G`, where `G` is defined by the clause `G :- d,B` and `d` is a constraint entailed by `c`, and then specializing `G`, instead of `H`. Most generalization strategies are based on *widening* operators like the ones introduced in the field of abstract interpretation [13]. Unfortunately, the use of generalization can lead to a loss of precision, that is, to a specialized set of verification conditions which we are neither able to prove satisfiable nor to prove unsatisfiable because of the presence of recursive clauses. Precision can be improved by iterating the specialization process and alternating the propagation of the constraints of the initial configuration (forward propagation) and of the error configuration (backward propagation).

Interpolation [14] is a technique that has been proposed to recover precision losses from abstractions, including those from widening (see, for instance, [2, 25]). Having found a spurious counterexample (that is, a false error detection), one can compute an interpolant formula that represents a set of states from which the spurious counterexample cannot be generated, and by combining widening with interpolation one can recover precision of the analysis by computing an overapproximation of the reachable states that rules out the counterexample. Interpolation (without any combination with widening) has also been used by several CLP verification techniques (see, for instance, [23, 24, 31, 33, 39]). However, to the best of our knowledge, no CLP-based verifier combines widening and interpolation in a nontrivial way.

Among the various CLP-based techniques, *Interpolating Horn Clause* (IHC) solvers [23] enhance the classical top-down (i.e., goal oriented) execution strategy for CLP with the use of interpolation. During top-down execution suitable interpolants are computed, and when the constraints accumulated during the execution of a goal imply an interpolant associated with that goal, then it is guaranteed that the goal will fail and the execution can be stopped. Thus, the use of interpolants can achieve termination in the presence of potentially infinite goal executions. Therefore, IHC solvers can be used to verify safety properties by showing that error conditions cannot be reached, even in the presence of infinite symbolic program executions.

In this paper we propose a CLP-based verification technique that combines widening and interpolation by composing in a modular way specialization-based verification with an IHC solver. Our technique first specializes the CLP clauses representing the verification conditions with respect to the constraints representing the initial configuration, hence deriving an equivalent set of CLP clauses that incorporate constraints derived by unfolding and generalization (including, in particular, loop invariants computed by widening). Next, the IHC solver is applied to the specialized CLP clauses, hence adding the effect of interpolation to the effect of widening. In the case where the interpolation-based analysis is not sufficient to check program correctness, the verification process proceeds by specializing the CLP clauses with respect to the constraints representing the error configuration, and then applying the IHC solver to the output program. Since program specialization preserves equivalence with respect to the property of interest, we can iterate the process consisting in alternating specialization (with respect to the initial or error configuration) and interpolation in the hope of eventually deriving CLP clauses for which we are able to prove correctness or incorrectness.

We have implemented our verification technique by integrating FTCLP, an IHC solver [23], within VeriMAP, a verifier based on iterated specialization [17]. The experimental results show that the integrated verifier improves the precision of each of its individual components.

2 An Introductory Example

Consider the C program in Figure 1(a) where the symbol $*$ represents a non-deterministic choice. A possible inductive invariant that proves the safety of this program is $x \geq 1 \wedge y \geq 0 \wedge x \geq y$.

Figure 1(b) shows the set of CLP clauses that represent the verification conditions for our verification problem. We will explain in Section 3 how to generate automatically those conditions, but very roughly each basic block in the C program is translated to a CLP clause and each assertion is negated. Note that the loop is translated into the recursive predicate new3. The key property of this translation is that the program in Figure 1(a) is safe iff the CLP predicate unsafe in Figure 1(b) is unsatisfiable, that is, all derivations of unsafe lead to failure.

	unsafe :- new2. new2 :- $X = 1, Y = 0$, new3$(X, Y, _)$. new3$(X, Y, _)$:- new4$(X, Y, _)$. new4(X, Y, C) :- $X1 = X + Y$, $Y1 = Y + 1, C \geq 1$, new3$(X1, Y1, C)$. new4(X, Y, C) :- $C \leq 0$, new6(X, Y, C). new6(X, Y, C) :- $D = 1, (X - Y) \geq 0$, new7(D, X, Y, C). new6(X, Y, C) :- $D = 0, (X - Y) \leq -1$, new7(D, X, Y, C). new7$(D, _, _, _)$:- $D = 0$.	unsafe :- new2. new2 :- $X = 1, Y = 0$, new4(X, Y). new4(X, Y) :- $X = 1, Y = 0$, $Y1 = 1, X1 = 1$, new5$(X1, Y1)$. new5(X, Y) :- $X = 1, Y \geq 0$, new8(X, Y). new8(X, Y) :- $X = 1, X1 = Y + 1$, $X1 \geq 1, Y1 = X1$, new9$(X1, Y1)$. new8(X, Y) :- $X = 1, Y \geq 0$, new10(X, Y). new10(X, Y):- $X = 1, Y \geq 2$. new9(X, Y) :- $X \geq 1, Y \geq 0$, new13(X, Y). new13(X, Y):- $X1 = X + Y, Y1 = Y + 1$, new9$(X1, Y1)$. new13(X, Y):- $X \geq 1, Y \geq 0$, new15(X, Y). new15(X, Y):- $X \geq 1, (X - Y) \leq -1$.

int $x = 1$;
int $y = 0$;
while $(*)$ {
$x = x + y$;
$y = y + 1$;
}
assert$(x >= y)$;		

(a)	(b)	(c)

Figure 1: (a) C program, (b) CLP clauses representing the verification conditions, and (c) CLP clauses after specialization.

Note that neither a top-down nor a bottom-up evaluation of the CLP program in Figure 1(b) will terminate, essentially due to the existence of the recursive predicate new3. To avoid this problem, our method performs a transformation based on *program specialization* [16]. We postpone the details of this transformation to Section 3, but the key property of this transformation is that it is *satisfiability-preserving*. That is, unsafe is satisfiable in the input program iff it is satisfiable in the transformed one. The benefits of this transformation come from the addition of new constraints (e.g., using *widening* techniques) that might make a top-down or bottom-up evaluation terminate. We show the result of this transformation in Figure 1(c). For clarity, we have removed from the transformation irrelevant arguments and performed some constraint simplifications.

The clause of predicate new2 encodes the initial conditions. The transformation introduces new definitions new4 and new8 after unrolling twice new3. The predicate new9 encodes the loop just after these two unrolls. Note that our transformation inserts two additional constraints $X \geq 1$ and $Y \geq 0$ that will play an essential role later. Although this transformed set of clauses might appear simpler to solve the predicate unsafe, it is still defined by recursive clauses with constrained facts. Even CLP systems with tabling [10, 12] will not terminate here.

Instead, consider an IHC solver following the approach of *Failure Tabled CLP* (FTCLP) [23]. In a nutshell, FTCLP augments tabled CLP [10, 12] by computing *interpolants* whenever a failed derivation

CLP constraints for Figure 1(b)	CLP constraints for Figure 1(c)
Π_1: ..., $X=1, Y=0$, $\underline{\mathsf{new3}(X,Y,\_)}$, $\mathsf{new4}(X,Y,\_)$, $X1=X+Y, Y1=Y+1$, $\underline{\mathsf{new3}(X1,Y1,\_)}$	$\Pi_{1'}$: $X=2, Y=2$, $\underline{\mathsf{new9}(X,Y)}$, $X\geq 1, Y\geq 0$, $\mathsf{new13}(X,Y)$, $X1=X+Y, Y1=Y+1$, $\underline{\mathsf{new9}(X1,Y1)}$
Π_2: ..., $X=1, Y=0$, $\mathsf{new3}(X,Y,\_)$, $\mathsf{new4}(X,Y,\_)$, $\mathsf{new6}(X,Y,\_)$, $X-Y\geq 0$, $D=1$, $\mathsf{new7}(D,X,Y,\_)$, $D=0$	$\Pi_{2'}$: $X=2, Y=2$, $\mathsf{new9}(X,Y)$, $X\geq 1, Y\geq 0$, $\mathsf{new13}(X,Y)$, $X\geq 1, Y\geq 0$, $\mathsf{new15}(X,Y)$ $X\geq 1, X-Y\leq -1$
Π_3: ..., $X=1, Y=0$, $\mathsf{new3}(X,Y,\_)$, $\mathsf{new4}(X,Y,\_)$, $\mathsf{new6}(X,Y,\_)$, $X-Y\leq -1$, $D=0$, $\mathsf{new7}(D,X,Y,\_)$, $D=0$	

Table 1: Executions of the CLP clauses from Figures 1(b)-(c)

is encountered during the top-down evaluation of the CLP program. If a call of predicate p has been fully executed and has not produced any solution then its interpolants denote the conditions under which the execution of p will always lead to failure. During the execution of a recursive predicate p the top-down evaluation might produce multiple copies of p, each one with different constraints c originated from the unwinding of the recursive clauses for p. Given two copies p' and p'' with constraints c' and c'', respectively, and being p'' a descendant of p', the tabling mechanism will check whether p'' is *subsumed* by p' (i.e., $c'' \sqsubseteq c'$). If this is the case, the execution can be safely stopped at p''. In tabling, rather than using c' it is common to use weaker constraints (in the logical sense) in order to increase the likelihood of subsumption. These weaker constraints are often called *reuse conditions*. The major difference with respect to standard tabled CLP is that FTCLP uses the interpolants as reuse conditions while tabled CLP uses *constraint projection*. A key insight is that when FTCLP is used in the context of verification its tabling mechanism using interpolants as reuse conditions resembles a verification algorithm using interpolants to produce *candidates* and keeping those which can be proven to be inductive invariants.

Coming back to our CLP program in Figure 1(b). If we run FTCLP on that set of clauses, the top-down execution of unsafe will not terminate. To understand why, let us focus on the predicate new3, initially with constraints $X=1, Y=0$. From new3 we reach new4 which has two clauses. From the execution of the first clause, we will reach again new3 (Π_1 in Table 1), but this time with constraints $X=1, Y=0, X1=X+Y, Y1=Y+1$ (we ignore constraints over C since they are irrelevant). To avoid an infinite loop, the tabling mechanism will freeze here its execution and backtrack to the nearest choice point which is the second clause for new4. From here, we will reach new6 which has also two clauses. The first clause for new6, executed by Π_2, has the constraint $X-Y\geq 0, D=1$ which is consistent with $X=1, Y=0, X1=X+Y, Y1=Y+1$. The execution follows new7 which will fail because the constraint $D=0$ cannot be satisfied. The second clause of new6 executed in Π_3 has no solution either because $D=0, X-Y\leq -1$ is also false. Very importantly, FTCLP will generate interpolants from these two failed derivations. The key interpolant is generated from the second clause of new6 (Π_3) which when propagated backwards to new3 is $X\geq Y$. Note that this is an essential piece of information, but unfortunately it is not enough for tabling to stop permanently since this constraint alone is not sufficient to subsume the descendants of new3. That is, $X\geq Y \wedge X1=X+Y \wedge Y1=Y+1 \not\models X1\geq Y1$. Therefore, FTCLP will unfreeze the execution of new3 and repeat the process. Unfortunately, FTCLP will not terminate.

Let us now consider the set of clauses in Figure 1(c). Let us repeat the same process running FTCLP on the transformed program. Let us focus on new9, which encodes the loop after two unrolls. For clarity of presentation the execution of new9, shown in $\Pi_{1'}$, starts with the constraints $X=2, Y=2$ which are produced by projecting the constraints accumulated from new2 to new9 onto X and Y. Since there is a cycle when $\mathsf{new9}(X1,Y1)$ is reached the tabling mechanism will freeze its execution. Note that the transformation inserted the constraints $X\geq 1$ and $Y\geq 0$. This was achieved by applying generalization

via widening during the unfolding of the clause. From the second clause of new13, shown in $\Pi_{2'}$, the top-down evaluation will eventually fail. Again, it will generate an interpolant that after backwards propagation to new9 is $X \geq Y$. Now, it can be proven that the descendant of new9 can be safely subsumed because $X \geq Y, X \geq 1, Y \geq 0, X1 = X + Y, Y1 = Y + 1 \models X1 \geq Y1$, and therefore, FTCLP will terminate proving that unsafe is unsatisfiable. The magic here is originated from the fact that the transformation produced the invariants $X \geq 0$ and $Y \geq 1$ (*widening*) while FTCLP produced the remaining part $X \geq Y$ (*interpolation*) which together form the desired safe inductive invariant.

Finally, it is worth mentioning that we do not claim that this program cannot be proven safe by other methods using more sophisticated interpolation algorithms (e.g. [4]) and/or combining with other techniques such as predicate abstraction (e.g., [24]). Instead, we would like to stress how we can have the same effect than a specialized algorithm using widening and interpolation in a much less intrusive and completely modular manner by exploiting the fact that both methods share the same CLP representation.

3 Verification based on Iterated Specialization and Interpolation

In order to show how our verification method based on iterated specialization and interpolation works, let us consider again the program shown in Fig. 1 (a). We want to prove safety of program P with respect to the initial configurations satisfying $\varphi_{init}(x,y) =_{def} x = 1 \wedge y = 0$, and the error configurations satisfying $\varphi_{error}(x,y) =_{def} x < y$. That is, we want to show that, starting from any values of x and y that satisfy $\varphi_{init}(x,y)$, after every terminating execution of program P, the new values of x and y do not satisfy $\varphi_{error}(x,y)$.

Our verification method consists of the following steps.

(*Step 1 - CLP Encoding*) First, we encode the safety verification problem using a CLP program I, called the *interpreter*, which defines a predicate `unsafe`. We have that program P is safe iff $I \not\models$ `unsafe`.

(*Step 2 - Verification Conditions Generation*) Then, by specializing I w.r.t. (a CLP encoding of) program P, we generate a set VC of CLP clauses representing the *verification conditions* for P. Specialization preserves safety, i.e., $I \models$ `unsafe` iff $VC \models$ `unsafe`.

(*Step 3 - Constraint Propagation*) Next, we apply CLP specialization again and we propagate the constraints occurring in the initial and error conditions, thereby deriving a new set VC' of verification conditions. Also this specialization step preserves safety, i.e., $VC \models$ `unsafe` iff $VC' \models$ `unsafe`. During this step, in order to guarantee termination, we generate inductive loop invariants by using generalization operators based on widening and convex hull [21].

(*Step 4 - Interpolating Verification*) We consider the verification conditions obtained after constraint propagation. If their satisfiability cannot be decided by simple syntactic checks (e.g., emptiness of the set of CLP clauses or absence of constrained facts), we apply the interpolating Horn Clause solver.

In the case where the IHC solver is not able to provide a definite answer, we *reverse* the program encoding the VC's by exchanging the direction of the transition relation and the role of the initial and error conditions, then we *iterate* the constraint propagation and satisfiability check (go to Step 3).

Of course, due to undecidability of safety, our verification method might not terminate. However, as experimentally shown in Section 4, the combination of program specialization and interpolating verification is successful in many examples and it is synergistic, in the sense that it improves over the use of program specialization and interpolating verification alone. We now describe in more detail the techniques applied in each step listed above.

3.1 Encoding safety problems of imperative programs using CLP

As already mentioned, the safety verification problem can be encoded as a CLP program I, called the interpreter. We now show, through an example, how to derive a set of CLP clauses that encode (i) the semantics of the programming language in which the program under verification is written, (i) the program under verification itself, and (iii) the proof rules for the considered safety property. The extension to the general case is straightforward.

Encoding the semantics of the programming language. The semantics of the imperative language can be encoded as a transition relation from any configuration of the imperative program to the next configuration, by using the predicate `tr`. Below we list the clauses of `tr` for (i) assignments (clause 1), (ii) conditionals (clauses 2 and 3), and (iii) jumps (clause 4).

```
1. tr(cf(cmd(L,asgn(X,expr(E))),Env), cf(cmd(L1,C),Env1)) :-
                    eval(E,Env,V), update(Env,X,V,Env1), nextlab(L,L1), at(L1,C).
2. tr(cf(cmd(L,ite(E,L1,L2)),Env), cf(cmd(L1,C),Env)) :- beval(E,Env),at(L1,C).
3. tr(cf(cmd(L,ite(E,L1,L2)),Env), cf(cmd(L2,C),Env)) :- beval(not(E),Env),at(L2,C).
4. tr(cf(cmd(L,goto(L)),Env), cf(cmd(L,C),Env)) :- at(L,C).
```

The term `cf(cmd(L,C),Env)` encodes the configuration consisting of the command `C` with label `L` and the environment `Env`. The predicate `eval(E,Env,V)` computes the value `V` of the expression `E` in the environment `Env`. The predicate `beval(E,Env)` holds if the Boolean expression `E` is true in the environment `Env`. The predicate `at(L,C)` binds to `C` the command with label `L`. The predicate `nextlab(L,L1)` binds to `L1` the label of the command that is written immediately after the command with label `L`. The predicate `update(Env,X,V,Env1)` updates the environment `Env` by binding the variable `X` to the value `V`, thereby constructing a new environment `Env1`.

Encoding the imperative program. The imperative program P is encoded by a set of constrained facts for the `at` predicate, as follows. First, we translate program P to the sequence of labelled commands $\ell_0 \ldots \ell_h$. Then, we introduce the CLP facts $\{5-9\}$ which encode those commands.

ℓ_0: `if` $(*)$ ℓ_1 `else` ℓ_h;	5. `at(0,ite(nondet,1,h)).`
ℓ_1: $x = x+y$;	6. `at(1,asgn(int(x),expr(plus(int(x),int(y))))).`
ℓ_2: $y = y+1$;	7. `at(2,asgn(int(y),expr(plus(int(y),int(1))))).`
ℓ_3: `goto` ℓ_0;	8. `at(3,goto(0)).`
ℓ_h: `halt;`	9. `at(h,halt).`

Encoding safety as reachability of configurations. Finally, we write the CLP clauses defining the predicate `unsafe` that holds if and only if there exists an execution of the program P that leads from an initial configuration to an error configuration.

```
10. unsafe :- initConf(X), reach(X).
11. reach(X) :- tr(X,X1), reach(X1).
12. reach(X) :- errorConf(X).
13. initConf(cf(cmd(0,C), [[int(x),X],[int(y),Y]])) :- X=1, Y=0, at(0,C).
14. errorConf(cf(cmd(h,halt), [[int(x),X],[int(y),Y]])) :- X<Y.
```

Note that in clauses 13 and 14, the second component of the configuration term `cf` encodes the environment as a list `[[int(x),X],[int(y),Y]]` that provides the bindings for the program variables x and y, respectively. The set of CLP clauses $\{1-14\}$ constitutes the interpreter I. By the correctness of the CLP Encoding [16] we have that program P is safe if and only if $I \not\models$ `unsafe`.

When the CLP program I is run by using the top-down, goal directed strategy usually employed by CLP systems, an execution of the goal `unsafe` corresponds to a forward traversal of the transition graph

(i.e., a traversal starting from the initial configuration). We can also provide an alternative definition of the `reach(X)` relation that would generate a backward traversal of the transition graph (i.e., a traversal starting from the error configuration). However, as shown in Section 3.5, the method alternates between the forward and backward directions, and the direction used by the first specialization is not very relevant in practice.

More in general, the specialization-based approach is to a large extent parametric with respect to the definition of the semantics of the programming language, as long as this is defined by a CLP program.

3.2 Generation of CLP Verification Conditions

The verification conditions associated with the given safety verification problem, are generated by performing a CLP specialization based on unfold/fold program transformations [19, 20]. During this step, we specialize the clauses defining `unsafe` in the interpreter I with respect to: (i) the clauses that define the predicate `tr`, and (ii) the clauses that encode the given program P. The output of this program specialization is the set of verification conditions for P: a set of CLP clauses that encode the unsafety of the imperative program, is equisatisfiable w.r.t. I, and contains no explicit reference to the predicate symbols used for encoding the transition relation and the program commands. This first step is performed also in other specialization-based techniques for program verification (see, for instance, [15, 37].).

In this paper we consider a C-like imperative programming language and proof rules for safety properties only. However, the specialization-based approach to the generation of verification conditions has the advantage of being modular w.r.t. (i) the semantics of the programming language in which the program under verification is written, and (ii) the logic used for specifying the property to be verified, and seems to be reasonably efficient in practice.

The verification conditions for the safety problem we are considering are shown in Fig. 1 (b) and have been briefly described in Section 2.

3.3 Constraint Propagation by CLP Transformation

Constraint propagation is achieved by a CLP specialization algorithm similar to the one used for the Verification Conditions Generation (Step 2). The main difference is that for constraint propagation we use a *generalization operator* based on widening and convex-hull that in many cases allows the discovery of useful program invariants [21].

The specialization algorithm starts off from a set VC of clauses that include $j \geq 1$ clauses defining the predicate `unsafe`:

$\texttt{unsafe :- } c_1\texttt{(X)}, p_1\texttt{(X)}, \ \ldots, \ \texttt{unsafe :- } c_j\texttt{(X)}, p_j\texttt{(X)}$

where $c_1\texttt{(X)}, \ldots, c_j\texttt{(X)}$ are constraints and $p_1\texttt{(X)}, \ldots, p_j\texttt{(X)}$ are atoms.

The specialization algorithm makes use of the *unfolding*, *definition introduction*, and *folding* transformation rules [19, 20].

The clauses for `unsafe` are unfolded, that is, they are transformed by applying the following unfolding rule: Given a clause C of the form $\texttt{H :- c, A}$, let $\{\texttt{K}_i \texttt{ :- c}_i\texttt{, Q}_i \mid i = 1, \ldots, m\}$ be the set of the (renamed apart) clauses in program VC such that, for $i = 1, \ldots, m$, `A` is unifiable with $\texttt{K}_i$ via the most general unifier ϑ_i and the constraint $(\texttt{c}, \texttt{c}_i)\,\vartheta_i$ is satisfiable. Then from clause C we derive the clauses:

$(\texttt{H :- c}, \texttt{c}_i, \texttt{Q}_i)\,\vartheta_i, \ \text{ for } i = 1, \ldots, m.$

Unfolding propagates the constraints by adding the constraints on atom A to the constraints of the atom $\texttt{Q}_i$ that occurs in the body of a clause unifying A. By unfolding we may derive a fact for `unsafe`, and hence the given program is unsafe. Alternatively, we may derive an empty set of clauses for `unsafe`,

and hence we infer safety. However, in most cases we will derive a nonempty set of clauses which are not constrained facts. In these cases the specialization algorithm introduces a set *Defs* of new predicate definitions, one for each clause which is not a constrained fact. More specifically, let E be a clause derived by unfolding of the form `H(X) :- e(X,X1), Q(X1)`, where `X` and `X1` are tuples of variables, `e(X,X1)` is a constraint, and `Q(X1)` is an atom. Then, the specialization algorithm introduces a new definition clause D: `newq(X1) :- g(X1), Q(X1)`, such that: (i) `newq` is a new predicate symbol, and (ii) `g(X1)` is a generalization of `e(X,X1)`, that is, `e(X,X1)` $\sqsubseteq$ `g(X1)` (for the first definition introduction step `g(X1)` is the *projection*, in the reals, of `e(X,X1)` onto the variables `X1`). Then clause E is folded using D, hence deriving the clause F: `H(X) :- e(X,X1), newq(X1)`. Note that, even if `g(X1)` is a generalization of `e(X,X1)`, clause E is *equisatisfiable* to the pair of clauses D and F.

The clauses in *Defs* are then processed similarly to the clauses for `unsafe`, by applying unfolding, adding new predicate definitions in *Defs*, and folding, and this unfolding-definition introduction-folding cycle is repeated until all clauses derived by unfolding can be folded using clauses introduced in *Defs* in a previous step, so that no new predicate definitions need be introduced.

The termination of the specialization algorithm depends on a strategy that controls the introduction of new definitions so that all clauses are eventually folded. To guarantee termination we use a generalization operator *Gen* which enjoys properties similar to the *widening operator* considered in the field of abstract interpretation [13]. In particular, given a clause E as above and a set *Defs* of predicate definitions, $Gen(E, Defs)$ is a clause `newq(X1) :- g(X1), Q(X1)`, such that `e(X,X1)` $\sqsubseteq$ `g(X1)` and, moreover, any sequence of applications of *Gen stabilizes*, that is, the following property holds. For any infinite sequence $E_1, E_2, \ldots$ of clauses, let $G_1, G_2, \ldots$ be a sequence of clauses constructed as follows: (1) $G_1 = Gen(E_1, \emptyset)$, and (2) for every $i > 0$, $G_{i+1} = Gen(E_{i+1}, \{G_1, \ldots, G_i\})$. Then there exists an index k such that, for every $i > k$, G_i is equal, modulo the head predicate name, to a clause in $\{G_1, \ldots, G_k\}$. Many concrete generalization operators have been defined in the CLP specialization literature (see, for instance, [21]), and we will consider two of them in our experiments of Section 4.

The correctness of the specialization algorithm directly follows from the fact that the transformation rules preserve the least model semantics [19]. Thus, given a set VC of CLP clauses (representing verification conditions), if VC' is the output of the specialization algorithm applied to VC, then $VC \models$ `unsafe` iff $VC' \models$ `unsafe`.

3.4 Interpolating solver

We now describe informally how the IHC solver considered in this paper works. For that, the concept of a derivation plays a key role. A *derivation step* is a transition from state $\langle G \mid C \rangle$ to state $\langle G' \mid C' \rangle$, written $\langle G \mid C \rangle \Rightarrow \langle G' \mid C' \rangle$, where G, G' are goals (sequences of literals that can be either atoms or constraints) and C, C' are *constraint stores* (the constraints accumulated during the derivation of a goal). A derivation step consists essentially of *unifying* some atom in the current goal G with the head of some clause and replacing G with the literals in the body of the matched clause producing a new goal G'. Moreover, new constraints can be added to the constraint store C producing a new constraint store C'. At any derivation step the constraint store C can be unsatisfiable, hence producing a *failed derivation*. A *derivation tree* for a goal G is a tree with states as nodes where each path corresponds to a possible derivation of G.

In order to prove that a goal (e.g., the predicate `unsafe`) is unsatisfiable, the solver tries to produce a finite derivation tree proving that the goal has no answers (i.e., all the derivations fail). If an answer is found then it represents a counterexample. To facilitate this process, each node in the tree is annotated with an interpolant producing at the end a *tree interpolant*. To achieve this, the solver computes a *path interpolant* from each failed derivation and then combines them. Informally, given a sequence of

formulas $F_1, \ldots, F_n$ (extracted from the constraint store at each state in the failed derivation) the sequence $I_0, \ldots, I_n$ is called a path interpolant if, for all $i \in [1, \ldots, n]$, we have $I_{i-1} \wedge F_i \models I_i$ (with $I_0 = true$ and $I_n = false$) and the variables of I_i are common to the variables of F_i and F_{i+1}. The interpolant associated with a node in the tree is the *conjunction* of the children's interpolants.

Even if the derivation tree is finite, its tree interpolant is very valuable since it can be used for pruning other redundant failed derivations.

A more interesting fact is how we can use interpolants to prove that the derivation of a goal will fail infinitely. We rely on the same principle followed by tabled CLP in order to subsume states in presence of infinite derivations. Whenever a cycle is detected its execution is frozen[2] to avoid running infinitely, and a backtracking to an ancestor choice point occurs. By repeating this, the execution of a goal will always terminate and a tree interpolant can be computed. After completion of a subtree, the tabling mechanism will attempt at proving that the state where the execution was frozen can be subsumed by any of its ancestors using an interpolant as the subsumption condition. If it fails then its execution is re-activated and the process continues. Of course, the execution might run forever. The subsumption test is described informally as follows. Let G_a and G_d be two atoms with the same functor and arity, where G_a is the head and G_d is the tail of a cycle. The symbols a and d refer to ancestor and descendant, respectively. Let $[\![p_1]\!], \ldots, [\![p_n]\!]$ be all the constraints originated from all paths $p_1, \ldots, p_n$ between G_a and G_d, and I_a be the interpolant computed for G_a. Then, we do not need to re-activate the execution of G_d if $\bigwedge_{1 \leq i \leq n} I_a \wedge [\![p_i]\!] \models I'_a$ (where I'_a is the interpolant I_a after proper renaming). This process is analogue to tabled CLP's *completion check*.

Consider again the transformed program in Figure 1(c). Let us focus on the execution of new9, which is a recursive predicate. Recall that new9 is reached after unwinding twice the loop. Therefore, before the execution of new9 the *constraint store* is $X = 2, Y = 2$, after constraint simplification. Its depth-first, left-to-right derivation tree is shown in Figure 2. Each oval node represents the call to a body atom and an edge denotes a derivation step. A failed derivation is marked with a (red) "cross" symbol. Note that there is no successful derivation, otherwise the program would be unsafe.

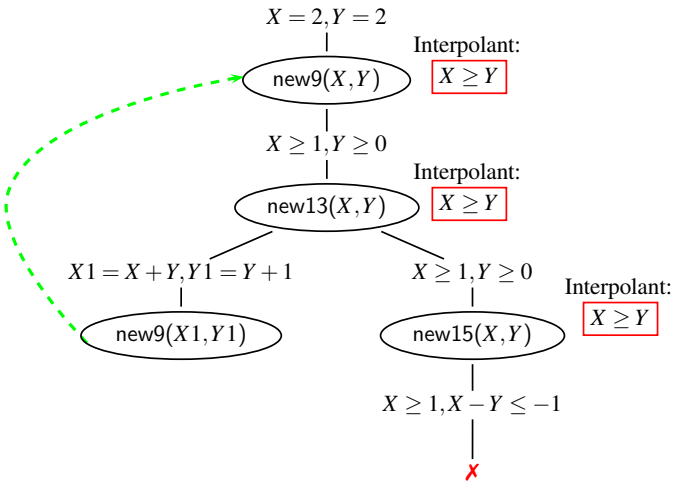

Figure 2: The derivation tree for goal $X = 2, Y = 2, \mathsf{new9}(X, Y)$ where new9 is a predicate defined in the CLP program of Figure 1(c)

[2] The freeze of an execution can be done in several ways. The algorithm described in [23] performs a counter instrumentation similar to [34] in order to make finite the execution and produce interpolants.

The leftmost derivation is frozen when the atom $\mathsf{new9}(X1,Y1)$ is encountered. Then, the execution backtracks to $\mathsf{new13}(X,Y)$ and activates the rightmost derivation, which fails. We compute the interpolant for this derivation and we annotate the atom $\mathsf{new9}(X,Y)$ with the constraint $X \geq Y$. At this point, we visit again $\mathsf{new9}(X1,Y1)$ in order to perform the subsumption test: $X \geq Y \wedge X \geq 1 \wedge Y \geq 0 \wedge X1 = X+Y \wedge Y1 = Y+1 \models X1 \geq Y1$. This entailment holds, and therefore the execution can safely stop proving that the goal $\mathsf{new9}(X,Y)$ is unsatisfiable. Note that if the transformation had not added the constraints $X \geq 1, Y \geq 0$ the subsumption test would have failed and the execution would have run forever.

Finally, the criteria used for stopping the IHC solver is currently based on a timeout. Of course, due to undecidability reasons, there is no method that can decide whether the IHC solver will eventually stop finding a safe inductive invariant. However, there might be cases where by inspecting the interpolants we could guess that it is not likely for the IHC solver to stop in a reasonable amount of time. In these cases, it is desirable to switch to the next transformation phase instead of waiting until the timeout expires. In the future, we would like to investigate this problem.

3.5 CLP reversal

In the case where the IHC solver is not able to check (within a given amount of time) whether `unsafe` holds or not, the verification method returns to Step 3 and propagates the constraints by first inverting the roles of the initial and error configurations. Thus, at each iteration of the method, verification switches from forward propagation (of the constraints of the initial configuration) to backward propagation (of the constraints of the error configuration), or vice versa, having also strengthened the constraints of the initial and error configurations due to previous specializations. This switch is achieved by a CLP transformation called *Reversal* [16].

CLP Reversal transforms the set $VC1 = \{1,2,3\}$ of CLP clauses into the set $VC2 = \{4,5,6\}$.

1. `unsafe :- a(U), r1(U).`
2. `r1(U) :- c(U,V), r1(V).`
3. `r1(U) :- b(U).`
4. `unsafe :- b(U), r2(U).`
5. `r2(V) :- c(U,V), r2(U).`
6. `r2(U) :- a(U).`

The Reversal transformation can be generalized to any number of clauses and predicates, and preserves safety in the sense that $VC1 \models$ `unsafe` iff $VC2 \models$ `unsafe`.

4 Experimental Evaluation

The verification method presented in Section 3 has been implemented by combining VeriMAP [17] and FTCLP [23]. The verification process is controlled by VeriMAP, which is responsible for the orchestration of the following components: (i) a *translator*, based on the C Intermediate Language (CIL) [36], which translates a given verification problem (i.e., the C program together with the initial and error configurations) into a set of CLP program, (ii) a *specializer* for CLP programs, based on the MAP transformation system [1], which generates the verification conditions (VCs) and applies the iterated specialization strategy, and (iii) an *IHC solver* (*IHCS*), implemented by the FTCLP tool.

We have performed an experimental evaluation on a set of benchmarks consisting of 216 verification problems (179 of which are safe, and the remaining 37 are unsafe). Most problems have been taken from the repositories of other tools such as DAGGER [25] (21 problems), TRACER [32] (66 problems), InvGen [27] (68 problems), and also from the TACAS 2013 Software Verification Competition [7] (52 problems). The size of the input programs ranges from a dozen to about five hundred lines of code. The source code of all the verification problems is available at `http://map.uniroma2.it/VeriMAP/hcvs/`.

	FTCLP	VeriMAP$_M$	VeriMAP$_M$ + FTCLP	VeriMAP$_{PH}$	VeriMAP$_{PH}$ + FTCLP
answers	116	128	160	178	182
crashes	5	0	2	0	0
timeouts	95	88	54	38	34
total time	12470.26	11285.77	9714.41	5678.09	6537.17
average time	107.50	88.17	60.72	31.90	35.92

Table 2: Verification results using VeriMAP, FTCLP, and the combination of VeriMAP and FTCLP. The timeout limit is two minutes. Times are in seconds.

The program verifier has been configured to execute the following process:

$$\mathit{Specialize}_{\mathit{Remove}};\ \mathit{Specialize}_{\mathit{Prop}};\ \mathit{IHCS};\ \big(\mathit{Reverse};\ \mathit{Specialize}_{\mathit{Prop}};\ \mathit{IHCS}\big)^*$$

After having translated the verification problem P into CLP, the verifier: (i) generates the verification conditions for P by applying the $\mathit{Specialize}_{\mathit{Remove}}$ procedure (Section 3.2), (ii) propagates the constraints that represent the initial configurations by executing the $\mathit{Specialize}_{\mathit{Prop}}$ procedure (Section 3.3), and (iii) runs the *IHC* solver (Section 3.4). If the solvability of the CLP clauses can be decided the verifier stops. Otherwise, the verifier calls the *Reverse* procedure that interchanges the roles of the initial and error configurations (Section 3.5), and calls $\mathit{Specialize}_{\mathit{Prop}}$ again. The $\big(\mathit{Reverse};\ \mathit{Specialize}_{\mathit{Prop}};\ \mathit{IHCS}\big)$ sequence might repeat forever unless the specializer is able to generate a set of CLP clauses that *IHCS* can either prove to be solvable or prove to be unsolvable.

All experiments have been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under the GNU Linux operating system Ubuntu 12.10 (64 bit, kernel version 3.2.0-57). A timeout limit of two minutes has been set for each verification problem.

Table 2 summarizes the verification results obtained by the VeriMAP and the FTCLP tools executed separately (first, second and fourth columns) and the combination of both tools (third and fifth columns). When VeriMAP is executed without the help of FTCLP, the analysis described in [16] is used in place of the IHC solver. In the columns labeled by VeriMAP$_M$ and VeriMAP$_{PH}$ we have reported the results obtained by using the VeriMAP system with the generalization operator Gen_M (*monovariant generalization*[3] using widening only) and Gen_{PH} (*polyvariant generalization*[4] using widening and convex hull), respectively. Row 1 reports the total number of definite answers (correctly asserting either program *safety* or *unsafety*). Row 2 reports the number of tool crashes. Row 3 reports the number of verification problems that could not be solved within the timeout limit of two minutes. Row 4 reports the total CPU time, in seconds, taken to run the whole set of verification tasks: it includes the time taken to produce answers and the time spent on tasks that timed out. Finally, row 5 reports the average time needed to produce a definite answer, which is obtained by dividing the total time by the number of answers.

The results in Table 2 show that the combination of VeriMAP and FTCLP, by exploiting the synergy of widening and interpolation, improves the performance of both tools whenever executed separately. In particular, we have that the best performance is achieved by the combination VeriMAP$_{PH}$ + FTCLP where the process is able to provide an answer for 182 programs out of 216 (84.26%).

Table 3 summarizes the results obtained at the end of each of the first five iterations of the verification process when VeriMAP is executed alone and combined with FTCLP. We observe that when VeriMAP is used in combination with FTCLP the number of iterations required to solve the verification problems is considerably reduced.

[3] All constrained atoms with the same predicate are generalized to the same new predicate.

[4] Constrained atoms with the same predicate can be generalized to different new predicates.

Iteration	VeriMAP$_M$	VeriMAP$_M$ + FTCLP	VeriMAP$_{PH}$	VeriMAP$_{PH}$ + FTCLP
1	74	119	104	136
2	45	38	54	34
3	7	2	10	5
4	2	1	8	3
5	0	0	2	4

Table 3: Number of definite answers computed by VeriMAP and by the combination of VeriMAP and FTCLP within the first five iterations.

5 Related Work

As Horn logic is becoming more popular for reasoning about properties of programs, the number of verifiers based on this logic has increased during recent years (e.g, [17, 24, 29, 32, 33, 39, 40]). Although they can differ significantly from each other, one possible classification is based on their use of interpolation (e.g., [24, 32, 33, 39]), Property Directed Reachability (PDR) [9] (e.g., [29]), or a combination of both (e.g., [40]). Unlike the above mentioned verifiers, VeriMAP [17] does not use interpolation and, as explained in previous sections, implements a transformation method based on widening techniques similar to the ones used in the field of abstract interpretation [13].

It should be noted that, with the exception of VeriMAP, abstract interpretation techniques are surprisingly less common than PDR and interpolation in Horn Clause verifiers. HSF [24] combines predicate abstraction with interpolation but no other abstract interpretations. TRACER [32] only uses abstract interpretation as a pre-processing step in order to inject invariants during the execution of the Horn Clauses. Therefore, to the best of our knowledge there is no Horn Clause verifier that combines abstract interpretation (apart from predicate abstraction) with interpolation in a nontrivial manner.

Several works (e.g., [2, 25, 26, 41]) have focused on how to refine abstract interpretations different from predicate abstraction outside the scope of Horn Clause verifiers. [26, 41] focus on how to recover from the losses of widening by using specific knowledge of polyhedra. DAGGER [25] tackles in a more general way the imprecision due to widening by proposing the "interpolated widen" operator which refines the abstract state after widening using interpolation. [2] proposes another algorithm called VINTA which can also refine precision losses from widening, but it relies heavily on the use of an abstract domain that can represent efficiently disjunction of abstract states. However, efficient disjunctive abstract domains are rare and it is well known that the design of precise widening operators is far from easy. UFO [3, 4] is a framework for combining CEGAR methods based on over-and-under approximations which is parameterized by a *post* operator. The post operator is used during the unwinding of the control flow graph. If an error is found then Craig interpolation is used to refine the abstraction. Although in principle the post operator could perform an arbitrary abstraction the refinement described in [3, 4] assume heavily that predicate abstraction is used. If other abstractions were used it is not clear at all how to refine them. Moreover, unlike DAGGER and VINTA, during the unwinding of the control flow graph no abstract joins are performed, and thus we may consider UFO as another CEGAR method based on interpolation.

The approach followed by DAGGER is probably the most closely related to ours. If DAGGER finds a spurious counterexample due to widening losses, it must be the case that $(A \sqcup B) \sqcap E = \bot$ but $(A \nabla B) \sqcap E \neq \bot$, where A is the abstract state before starting the execution of a loop, B is the abstract state after executing the backedge of the loop, and E is an abstract state that leads to an error. The idea behind "interpolated widen" is to replace ∇ with ∇_I, where ∇_I is an instance of the widening up-to [28]. The key

property of the ∇_I operator is that it preserves the desirable properties of widening while excludes from the abstract state $A \nabla_I B$ the spurious counterexample (and possibly others) denoted by the interpolant I (i.e., $(A \nabla_I B) \sqcap E = \bot$). Our transformation phase performs widening during the generalization step, while the IHC solver generates interpolants in order to discover more program invariants. We believe this combination can be seen as a version of the ∇_I operator. The main difference is that our method can obtain an effect similar to combining widening with interpolation without the enormous effort of implementing a new verifier from scratch.

6 Conclusions and Future Work

In this paper we have presented some preliminary results obtained by integrating an Interpolating Horn Clause solver (FTCLP) with an Iterated Specialization tool (VeriMAP). The experimental evaluation confirms that such an integration is effective in practice, as discussed in Section 4.

The fact that both tools use CLP as a representation formalism for the verification conditions, together with the modular design of VeriMAP, allowed us a very clean and painless integration with FTCLP. As a result, we can achieve the effect of combining abstract interpretation with interpolation without having to design and implement a custom verifier. We believe this modular combination is valuable by itself, since based on the experience, one could implement a custom verifier or simply apply the method described here if the performance is adequate.

In this preliminary work, we have used the IHC solver mainly as a black-box and although the gains are promising they are somewhat limited. As future work, we would like to combine these tools in more *synergistic* ways. We believe that the integration can be improved in several ways.

First, when FTCLP is not able to produce a solution within the considered timeout limit, it would be useful to leverage the partial information it discovers and integrate it in the transformed program, with the aim of improving the subsequent unfold/fold transformation steps. For example, during its execution FTCLP might discover that some subtrees rooted in some goal cannot lead to an answer. Thus, the corresponding predicate can be considered useless and its clauses can be removed from the specialized program before the next iteration starts.

Another observation is that FTCLP generates for each predicate p an interpolant that represents an over-approximation of the original constraint store that preserves the unsolvability of p. It would be interesting to study how these interpolants can be used to refine the generalization step performed during the unfold/fold transformation, with the objective of preserving the branching structure of the symbolic evaluation tree, as indicated in [15], and preventing the introduction of spurious paths.

Finally, another possible direction for future work regards the use of interpolation *during* the transformation process in order to make more efficient the unfold/fold process. While this appears to be a very promising direction it raises some issues related to the termination of the transformation process itself, which deserve further study.

Acknowledgments

We would like to thank the anonymous referees for their helpful and constructive comments. This work has been partially supported by the Italian National Group of Computing Science (GNCS-INDAM).

References

[1] *The MAP transformation system*. Available at `www.iasi.cnr.it/~proietti/system.html`.

[2] Aws Albarghouthi, Arie Gurfinkel & Marsha Chechik (2012): *Craig Interpretation*. In: *Proceedings of SAS*, pp. 300–316, doi:10.1007/978-3-642-33125-1_21.

[3] Aws Albarghouthi, Arie Gurfinkel & Marsha Chechik (2012): *From Under-Approximations to Over-Approximations and Back*. In: *Proceedings of TACAS*, pp. 157–172, doi:10.1007/978-3-642-28756-5_12.

[4] Aws Albarghouthi, Yi Li, Arie Gurfinkel & Marsha Chechik (2012): *Ufo: A Framework for Abstraction- and Interpolation-Based Software Verification*. In: *Proceedings of CAV*, pp. 672–678, doi:10.1007/978-3-642-31424-7_48.

[5] Elvira Albert, Miguel Gómez-Zamalloa, Laurent Hubert & Germán Puebla (2007): *Verification of Java Bytecode Using Analysis and Transformation of Logic Programs*. In: *Proceedings of PADL*, pp. 124–139, doi:10.1007/978-3-540-69611-7_8.

[6] Roberto Bagnara, Patricia M. Hill & E.nea Zaffanella (2008): *The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems*. *Science of Computer Programming* 72(1-2), pp. 3–21, doi:10.1016/j.scico.2007.08.001.

[7] Dirk Beyer (2013): *Competition on Software Verification - (SV-COMP)*. In: *Proceedings of TACAS*, pp. 594–609, doi:10.1007/978-3-642-36742-7_43.

[8] Nikolaj Bjørner, Kenneth McMillan & Andrey Rybalchenko (2012): *Program Verification as Satisfiability Modulo Theories*. In: *Proceedings of SMT*, pp. 3–11.

[9] Aaron R. Bradley (2011): *SAT-Based Model Checking without Unrolling*. In: *Proceedings of VMCAI*, LNCS 6538, Springer, pp. 70–87, doi:10.1007/978-3-642-18275-4_7.

[10] P. Chico de Guzmán, M. Carro, M. V. Hermenegildo & P. J. Stuckey (2012): *A General Implementation Framework for Tabled CLP*. In: *Proceedings of FLOPS*, pp. 104–119, doi:10.1007/978-3-642-29822-6_11.

[11] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma & Roberto Sebastiani (2013): *The MathSAT5 SMT Solver*. In Nir Piterman & Scott Smolka, editors: *Proceedings of TACAS*, *LNCS* 7795, Springer, doi:10.1007/978-3-642-36742-7_7.

[12] Philippe Codognet (1995): *A Tabulation Method for Constraint Logic Programming*. In: *Symposium and Exhibition on Industrial Applications of Prolog*.

[13] Patrick Cousot & Radhia Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: *Proceedings of POPL*, ACM, pp. 238–252, doi:10.1145/512950.512973.

[14] W. Craig (1957): *Linear Reasoning: A New Form of the Herbrand-Gentzen Theorem*. *Journal of Symbolic Logic* 22(3), pp. 250–268, doi:10.2307/2963593.

[15] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2013): *Specialization with Constrained Generalization for Software Model Checking*. In: *Proceedings of LOPSTR*, LNCS 7844, Springer, pp. 51–70, doi:10.1007/978-3-642-38197-3_5.

[16] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2014): *Program Verification via Iterated Specialization*. *Science of Computer Programming (Special Issue on PEPM 2013)*, doi:10.1016/j.scico.2014.05.017.

[17] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2014): *VeriMAP: A Tool for Verifying Programs through Transformations*. In: *Proceedings of TACAS*, pp. 568–574, doi:10.1007/978-3-642-54862-8_47.

[18] Gregory J. Duck, Joxan Jaffar & Nicolas C. H. Koh (2013): *Constraint-Based Program Reasoning with Heaps and Separation*. In: *Proceedings of CP*, LNCS 8124, Springer, pp. 282–298, doi:10.1007/978-3-642-40627-0_24.

[19] Sandro Etalle & Maurizio Gabbrielli (1996): *Transformations of CLP Modules*. *Theoretical Computer Science* 166(1&2), pp. 101–146, doi:10.1016/0304-3975(95)00148-4.

[20] Fabio Fioravanti, Alberto Pettorossi & Maurizio Proietti (2000): *Automated strategies for specializing constraint logic programs*. In: *Proceedings of LOPSTR*, doi:10.1007/3-540-45142-0_8.

[21] Fabio Fioravanti, Alberto Pettorossi, Maurizio Proietti & Valerio Senni (2013): *Generalization Strategies for the Verification of Infinite State Systems*. *Theory and Practice of Logic Programming* 13(2), pp. 175–199, doi:10.1017/S1471068411000627.

[22] John P. Gallagher & Bishoksan Kafle (2014): *Analysis and Transformation Tools for Constrained Horn Clause Verification*. In: *Proceedings of ICLP (to appear)*.

[23] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard & Peter J. Stuckey (2013): *Failure tabled constraint logic programming by interpolation*. *Theory and Practice of Logic Programming* 13(4-5), pp. 593–607, doi:10.1017/S1471068413000379.

[24] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea & Andrey Rybalchenko (2012): *Synthesizing Software Verifiers from Proof Rules*. In: *Proceedings of PLDI*, pp. 405–416, doi:10.1145/2254064.2254112.

[25] Bhargav S. Gulavani, Supratik Chakraborty, Aditya V. Nori & Sriram K. Rajamani (2008): *Automatically Refining Abstract Interpretations*. In: *Proceedings of TACAS*, pp. 443–458, doi:10.1007/978-3-540-78800-3_33.

[26] Bhargav S. Gulavani & Sriram K. Rajamani (2006): *Counterexample Driven Refinement for Abstract Interpretation*. In: *Proceedings of TACAS*, pp. 474–488, doi:10.1007/11691372_34.

[27] Ashutosh Gupta & Andrey Rybalchenko (2009): *InvGen: An Efficient Invariant Generator*. In: *Proceedings of CAV*, pp. 634–640, doi:10.1007/978-3-642-02658-4_48.

[28] Nicolas Halbwachs, Yann-Erick Proy & Patrick Roumanoff (1997): *Verification of Real-Time Systems using Linear Relation Analysis*. *Formal Methods in System Design* 11(2), pp. 157–185, doi:10.1023/A:1008678014487.

[29] Krystof Hoder, Nikolaj Bjørner & Leonardo Mendonça de Moura (2011): *μZ - An Efficient Engine for Fixed Points with Constraints*. In: *Proceedings of CAV*, pp. 457–462, doi:10.1007/978-3-642-22110-1_36.

[30] J. Jaffar & J. Lassez (1987): *Constraint Logic Programming*. In: *Proceedings of POPL*, pp. 111–119, doi:10.1145/41625.41635.

[31] J. Jaffar, A. E. Santosa & R. Voicu (2009): *An Interpolation Method for CLP Traversal*. In: *Proceedings of CP*, pp. 454–469, doi:10.1007/978-3-642-04244-7_37.

[32] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas & Andrew E. Santosa (2012): *TRACER: A Symbolic Execution Tool for Verification*. In: *Proceedings of CAV*, pp. 758–766, doi:10.1007/978-3-642-31424-7_61.

[33] Kenneth McMillan & Andrey Rybalchenko (2013): *Computing Relational Fixed Points using Interpolation*. Technical Report, MSR-TR-2013-6.

[34] Kenneth L. McMillan (2010): *Lazy Annotation for Program Testing and Verification*. In: *Proceedings of CAV*, pp. 104–118, doi:10.1007/978-3-642-14295-6_10.

[35] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Proceedings of TACAS*, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[36] George C. Necula, Scott McPeak, Shree Prakash Rahul & Westley Weimer (2002): *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*. In: *Proceedings of CC*, pp. 213–228, doi:10.1007/3-540-45937-5_16.

[37] J. C. Peralta, J. P. Gallagher & H. Saglam (1998): *Analysis of Imperative Programs through Analysis of Constraint Logic Programs*. In: *Proceedings of the 5th International Symposium on Static Analysis, SAS '98*, Lecture Notes in Computer Science 1503, Springer, pp. 246–261, doi:10.1007/3-540-49727-7_15.

[38] Germán Puebla, Elvira Albert & Manuel V. Hermenegildo (2006): *Abstract Interpretation with Specialized Definitions*. In: *Proceedings of SAS*, pp. 107–126, doi:10.1007/11823230_8.

[39] Philipp Rümmer, Hossein Hojjat & Viktor Kuncak (2013): *Disjunctive Interpolants for Horn-Clause Verification*. In: *Proceedings of CAV*, pp. 347–363, doi:10.1007/978-3-642-39799-8_24.

[40] Yakir Vizel & Arie Gurfinkel (2014): *Interpolating Property Directed Reachability*. In: *Proceedings of CAV*, doi:10.1007/978-3-319-08867-9_17.

[41] Chao Wang, Zijiang Yang, Aarti Gupta & Franjo Ivancic (2007): *Using Counterexamples for Improving the Precision of Reachability Computation with Polyhedra*. In: *Proceedings of CAV*, pp. 352–365, doi:10.1007/978-3-540-73368-3_40.