An Interpolation Method for CLP Traversal

Joxan Jaffar, Andrew E. Santosa, and Răzvan Voicu

School of Computing, National University of Singapore
{joxan,andrews,razvan}comp.nus.edu.sg

Abstract. We consider the problem of exploring the search tree of a CLP goal in pursuit of a target property. Essential to such a process is a method of tabling to prevent duplicate exploration. Typically, only actually traversed goals are memoed in the table. In this paper we present a method where, upon the successful traversal of a subgoal, a *generalization* of the subgoal is memoed. This enlarges the record of already traversed goals, thus providing more pruning in the subsequent search process. The key feature is that the abstraction computed is guaranteed not to give rise to a spurious path that might violate the target property.

A driving application area is the use of CLP to model the behavior of other programs. We demonstrate the performance of our method on a benchmark of program verfication problems.

1 Introduction

In this paper we present a general method for optimizing the traversal of general search trees. The gist of the method is backward-learning: proceeding in a depth-first manner, it discovers an *interpolant* from the completed exploration of a subtree. The interpolant describes properties of a more general subtree which, importantly, preserves the essence of the original subtree with respect to a *target property*. We show via experiments that often, the generalized tree is considerably more general than the original, and therefore its representation is considerably smaller.

Our method was originally crafted as a means to optimize the exploration of states in computation trees, which are used as a representation of program behaviour in program analysis and verification. Such a representation can be symbolic in that a single node represents not one but a possibly infinite set of concrete program states or traces. The importance of a computation tree stems from the fact that it can represent a *proof* of some property of the program. Building such a tree in fact is an instance of a search problem in the sense of Constraint Programming, see eg. [1], and viewed as such, the problem of state-space exploration essentially becomes the problem of traversing a search tree. In this circumstance, the target property can simply be a predicate, corresponding to a *safety property*. Or it can be something more general, like the projection onto a set of distinguished variables; in this example, preserving the target property would mean that the values of these variables remain unchanged.

More concretely, consider a CLP derivation tree as a decision tree where a node has a conjunction of formulas symbolically representing a set of states. Its successor node has an incrementally larger conjunction representing a new decision. Suppose the target nodes are the terminal nodes. During a depth-first traversal, whenever a path in the tree is traversed completely, we compute an *interpolant* at the target node. Where F denotes the formula in this node and T denotes the target property, this interpolant is a formula F' such that $F \models F'$ and $F' \models T$. (Failure is reported if no such F' can be found, ie: that $F \not\models T$.) Any such F' not only establishes that this node satisfies the target property, but also establishes that a generalization of F will also suffice. This interpolant can now be propagated back along the same path to ancestor states resulting in their possible generalizations. The final generalization of a state is then the conjunction of the possible generalizations of derivation paths that emanate from this state.

One view of the general method is that it provides an enhancement to the general method of *tabling* which is used in order to avoid duplicate or redundant search. In our case, what is tabled is not the encountered state itself, but a *generalization* of it.

The second part of the paper presents a specific algorithm for both computing and propagating interpolants throughout the search tree. The essential idea here is to consider the formulas describing subgoals as syntactic entities, and then to use serial constraint replacement successively on the individual formulas, starting in chronological order of the introduction of the formulas. In this way, we achieve efficiency and can still obtaining good interpolants.

1.1 Related Work

Tabling for logic programming is well known, with notable manifestation in the SLG resolution [2, 3] which is implemented in the XSB logic programming system [4]. As mentioned, we differ by tabling a generalization of an encountered call.

Though we focus on examples of CLP representing other programs, we mentioned that we have employed an early version of the present ideas for different problems. In [5], we enhanced the dynamic programming solving of resource-constrained shortest path (RCSP) problems. This kind of example is similar to a large class of combinatorial problems.

Our interpolation method is related to various no-good learning techniques in CSP [6] and conflict-driven and clause learning techniques in SAT solving [7–10]. These techniques identify subsets of the minimal *conflict set* or *unsatisfiable core* of the problem at hand w.r.t. a subtree. This is similar to our use of interpolation, where we generalize a precondition “just enough” to continue to maintain the verified property.

An important alternative method for proving safety of programs is translating the verification problem into a Boolean formula that can then be subjected to SAT or SMT solvers [11–16]. In particular, [8] introduces bounded model checking, which translates k-bounded reachability into a SAT problem. While practically efficient in case when the property of interest is violated, this approach is in fact incomplete, in the sense that the state space may never be fully explored. An improvement is presented in [17], which achieves unbounded model checking by using interpolation to successively refine an abstract transition relation that is then subjected to an external bounded model checking procedure. Techniques for generating interpolants, for use in state-of-the-art SMT solvers, are presented in [18]. The use of interpolants can also be seen in the area of theorem-proving [19].

In the area of program analysis, our work is related to various techniques of abstract interpretation, most notably *counterexample-guided abstraction refinement* (*CE-*

```
⟨0⟩ if (∗) then x := x+1
⟨1⟩ if (y ≥ 1) then x := x+2
⟨2⟩ if (y < 1) then x := x+4⟨3⟩
```

(a)

$p_0(X,Y)$:- $p_1(X,Y)$.
$p_0(X,Y)$:- $p_1(X',Y), X' = X+1$.
$p_1(X,Y)$:- $p_2(X,Y), Y < 1$.
$p_1(X,Y)$:- $p_2(X',Y), X' = X+2, Y \geq 1$.
$p_2(X,Y)$:- $p_3(X,Y), Y \geq 1$.
$p_2(X,Y)$:- $p_3(X',Y), X' = X+4, Y < 1$.
$p_3(X,Y,X,Y)$.

(b)

Fig. 1. A Program and Its CLP Model

$p_0(X)$:- $p_1(X'), X' = X+1$.
$p_0(X)$:- $p_1(X)$.
$p_1(X)$:- $p_2(X'), X' = X+2$.
$p_1(X)$:- $p_2(X)$.
$p_2(X)$:- $p_3(X'), X' = X+4$.
$p_2(X)$:- $p_3(X)$.
$p_3(X)$.

(a)

$p_0(I,N,X,Y)$:- $p_1(I,N,X,Y), \varphi(X)$.
$p_0(I,N,X,Y)$:- $p_1(I,N,X,Y), \neg\varphi(X)$.
$p_1(I,N,X,Y)$:- $p_2(I,N,X,Y), I \leq N$.
$p_1(I,N,X,Y)$:- $p_4(I,N,X,Y), I > N$.
$p_2(I,N,X,Y)$:- $p_3(I,N,X,Y'), Y' = Y \times (-Y)$.
$p_3(I,N,X,Y)$:- $p_1(I',N,X,Y), I' = I+1$.
$p_4(I,N,X,Y)$.

(b)

Fig. 2. CLP Programs

GAR) [20–22], which perform successive refinements of the abstract domain to discover the right abstraction to establish a safety property of a program. An approach that uses interpolation to improve the refinement mechanism of CEGAR is presented in [22, 23]. Here, interpolation is used to improve over the method given in [21], by achieving better locality of abstraction refinement. Ours differ from CEGAR formulations in some important ways: first, instead of *refining*, we *abstract* a set of (concrete) states. In fact, our algorithm abstracts a state after the computation subtree emanating from the state has been completely traversed, and the abstraction is constructed from the interpolations of the constraints along the paths in the subtree. Thus a second difference: our algorithm interpolates a *tree* as opposed to a *path*. More importantly, our algorithm does not traverse *spurious paths*, unlike abstract interpretation. We shall exemplify this difference in a comparison with BLAST [24] in Section 6.

2 The Basic Idea

Our main application area is the state-space traversal of imperative programs. For this, we model imperative programs in CLP. Such modeling has been presented in various works [25–27] and is informally exemplified here. Consider the imperative program of Fig. 1 (a). Here the $*$ denotes a condition of nondeterministic truth value. We also augment the program with program points enclosed in angle brackets. The CLP model of the same program is shown in Fig. 1 (b).

To exemplify our idea, let us first consider a simpler CLP program of Fig. 2 (a), which is a model of an imperative program. The execution of the CLP program results in the derivation tree shown in Fig. 3 (top). The derivation tree is obtained by reduction using p_i predicates in the CLP model. In Fig. 3 (top), we write a CLP goal $p_k(\tilde{X}), \varphi$ as

$\langle k\rangle, \varphi'$ where φ' is a simplification of φ by projecting on the variables $\tilde{X}$, and an arrow denotes a derivation step.

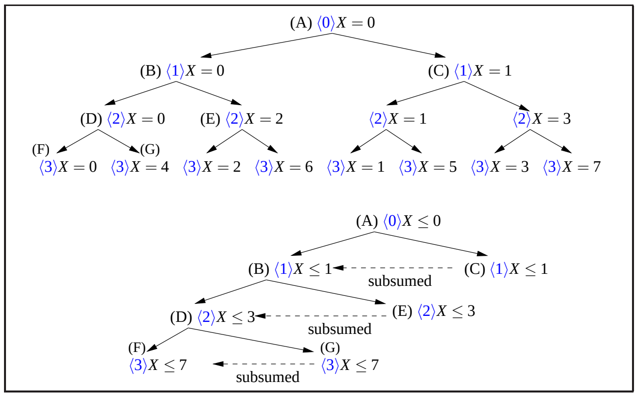


Fig. 3. Interpolation

Starting in a goal satisfying $X = 0$, all derivation sequences are depicted in Fig. 3 (top). Suppose the target property is that $X \leq 7$ at program point $\langle 3\rangle$. The algorithm starts reducing from (A) to (F). (F) is labelled with $X = 0$ which satisfies $X \leq 7$. However, a more general formula, say $X \leq 5$, would also satisfy $X \leq 7$. The constraint $X \leq 5$ is therefore an interpolant, since $X = 0$ implies $X \leq 5$, and $X \leq 5$ in turn implies $X \leq 7$. We could use such an interpolant to generalize the label of node (F). However, we would like to use as general an interpolant as possible, and clearly in this case, it is $X \leq 7$ itself. Hence, in Fig. 3 (bottom) we replace the label of (F) with $\langle 3\rangle$ $X \leq 7$. In this way, node (G) with label $\langle 3\rangle$ $X = 4$ (which has not yet been traversed) is now *subsumed* by node (F) with the new label (since $X = 4$ satisfies $X \leq 7$) and so (G) need not be traversed anymore. Here we can again generate an interpolant for (G) such that it remains subsumed by (F). The most general of these is $X \leq 7$ itself, which we use to label (G) in Fig. 3 (bottom).

We next use the interpolants of (F) and (G) to produce a generalization of (D). Our technique is to first compute candidate interpolants from the interpolants of (F) and (G), w.r.t. the reductions from (D) to (F) and from (D) to (G). The final interpolant of (D) is the conjunction of these candidate interpolants. In this process, we first rename the variables of (F) and (G) with their primed versions, such that (F) and (G) both have the label $X' \leq 7$. First consider the reduction from (D) to (F), which is in fact equivalent to a `skip` statement, and hence it can be represented as the constraint $X' = X$. It can be easily seen that the label $X = 0$ of (D) entails $X' = X \models X' \leq 7$. Here again we compute an interpolant. The interpolant here would be entailed by $X = 0$ and entails $X' = X \models X' \leq 7$. As interpolant, we choose $X \leq 7$[1].

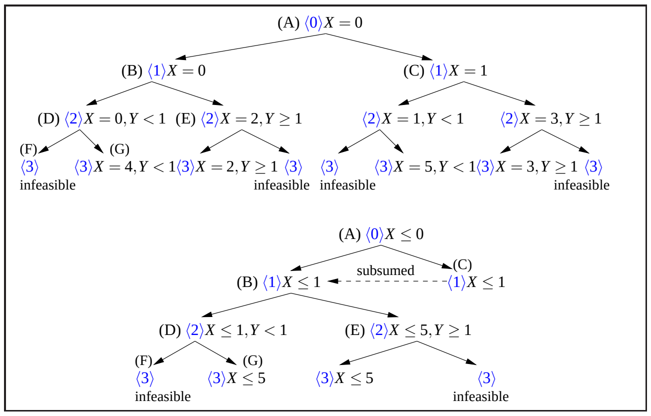
Similarly, considering the goal reduction (D) to (G) as the augmentation of the constraint $X' = X + 4$, we obtain a candidate interpolant $X \leq 3$ for (D). The final interpolant for (D) is the conjunction of all candidates, which is $X \leq 7 \wedge X \leq 3 \equiv X \leq 3$. We label (D) with this interpolant in Fig. 3 (bottom). In this way, (E) is now subsumed by (D), and its traversal for the verification of target property is not necessary.

[1] This interpolant corresponds to the *weakest precondition* [28] w.r.t. the statement $X := X$ and the target property $X \leq 7$, however, in general the obtained precondition need not be the weakest, as long as it is an interpolant.

We then generate an interpolant for (E) in the same way we did for (G). By repeating the process described above for other nodes in the tree, we obtain the smaller tree of Fig. 3 (bottom), which is linear in the size of the program. This tree represents the part of the symbolic computation tree that would *actually be traversed* by the algorithm. Hence, while the tree's size is exponential in the number of `if` statements, our algorithm prunes significant parts of the tree, speeding up the process.

2.1 With Infeasible Sequences

Now consider Fig. 1 (a) where there are *infeasible* sequences ending in goals with unsatisfiable constraint, which are depicted in Fig. 4 (top). Let the target property be $X \leq 5$ at point $\langle 3 \rangle$. A key principle of our re-labeling process is that it *preserves the infeasibility* of every derivation sequence. Thus, we must avoid re-labeling a node too generally, since that may turn infeasible paths into feasible ones. To understand why this is necessary, let us assume, for instance, re-labelling (E), whose original label is $X = 2, Y \geq 1$, into $X = 2$. This would yield $X = 6$ at point $\langle 3 \rangle$ (a previously unreachable goal), which no longer entails the target property $X \leq 5$.



(A) $\langle 0 \rangle X = 0$
(B) $\langle 1 \rangle X = 0$
(C) $\langle 1 \rangle X = 1$
(D) $\langle 2 \rangle X = 0, Y < 1$
(E) $\langle 2 \rangle X = 2, Y \geq 1$
$\langle 2 \rangle X = 1, Y < 1$
$\langle 2 \rangle X = 3, Y \geq 1$
(F) $\langle 3 \rangle$ infeasible
(G) $\langle 3 \rangle X = 4, Y < 1$
$\langle 3 \rangle X = 2, Y \geq 1$
$\langle 3 \rangle$ infeasible
$\langle 3 \rangle$ infeasible
$\langle 3 \rangle X = 5, Y < 1$
$\langle 3 \rangle X = 3, Y \geq 1$
$\langle 3 \rangle$ infeasible
(A) $\langle 0 \rangle X \leq 0$
(B) $\langle 1 \rangle X \leq 1$
subsumed
(C) $\langle 1 \rangle X \leq 1$
(D) $\langle 2 \rangle X \leq 1, Y < 1$
(E) $\langle 2 \rangle X \leq 5, Y \geq 1$
(F) $\langle 3 \rangle$ infeasible
(G) $\langle 3 \rangle X \leq 5$
$\langle 3 \rangle X \leq 5$
$\langle 3 \rangle$ infeasible

Fig. 4. Interpolation of Infeasible Sequences

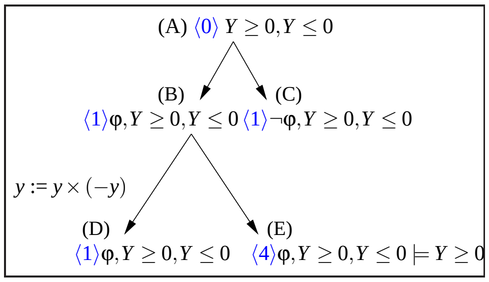
Next note that the path ending at (F) is also infeasible. Applying our infeasibility preservation principle, we keep (F) labeled with *false*, and therefore the only possible interpolant for (F) is *false* itself. This would produce the interpolant $Y < 1$ at (D) since this is the most general condition that preserves the infeasibility of (F). Note that here, $Y < 1$ is implied by the original label $X = 0, Y < 1$ of (D) and implies $Y \geq 1 \models$ *false*, which is the weakest precondition of *false* w.r.t. the negation of the `if` condition on the transition from (D) to (F).

Now consider (G) with $X = 4, Y < 1$ and note that it satisfies $X \leq 5$. (G) can be interpolated to $X \leq 5$. As before, this would produce the precondition $X \leq 1$ at (D). The final interpolant for (D) is the *conjunction* of $X \leq 1$ (produced from (G)) and $Y < 1$ (produced from (F)). In this way, (E) cannot be subsumed by (D) since its label does not satisfy both the old and the new labels of (D). Fortunately, however, after producing the interpolant for (B), the node (C) can still be subsumed.

In Section 5 we detail an efficient technique to generate interpolants called *serial constraint replacement*, which is based on constraint deletion and slackening. This technique is briefly explained next.

2.2 Loops

Our method assumes that the derivation tree is finite in the sense that there is no infinite sequence of distinct goals in a single path, although the tree may contain cyclic derivations. Here we discuss how we may compute an interpolant from cyclic derivations.



(A) ⟨0⟩ $Y \geq 0, Y \leq 0$
(B)
(C)
⟨1⟩$\varphi, Y \geq 0, Y \leq 0$
⟨1⟩$\neg\varphi, Y \geq 0, Y \leq 0$
$y := y \times (-y)$
(D)
(E)
⟨1⟩$\varphi, Y \geq 0, Y \leq 0$
⟨4⟩$\varphi, Y \geq 0, Y \leq 0 \models Y \geq 0$

Fig. 5. Loop Interpolation

Consider Fig. 2 (b) which is a program with cyclic derivation. The program contains some constraint φ. The derivation tree, where the initial goal is $Y \geq 0, Y \leq 0$, is shown in Fig. 5. The tree is finite because in (D), the second occurrence of point ⟨1⟩, is *subsumed* by the ancestor (B). This subsumption is enabled by a *loop invariant* made available by some external means at node (B), and which renders unnecessary the expansion of node (D) (the computation tree is“closed” at (D)).

In the spirit of the example, we now attempt to generalize node (B) in order to avoid having to traverse node (C) (which must be traversed if (B) were not generalized).

Let us first examine the path (A), (B), (D). The constraint φ can be removed from (B) so that the resulting goal remains a loop invariant. This is because φ is not related to the other variables. More importantly, φ is itself a loop invariant, and we come back to this later.

Next we attempt to remove the constraint $Y \leq 0$. The resulting goal at (B) now has the constraint $Y \geq 0$. But this goal is no longer invariant. That is, the computation tree at this new node (B) is such that the corresponding node (D) is *not* subsumed by (B). A similar situation would arise if we kept $Y \leq 0$ and deleted $Y \geq 0$.

The only possibility we have left is to check if we can remove *both* of $Y \leq 0$ and $Y \geq 0$. Indeed, that is possible, since the sequence (A), (B), (D), from which all constraints $\varphi, Y \leq 0, Y \geq 0$ are removed, is such that (B) subsumes (D). Indeed, in this case, the generalized (B) subsumes all the goals at point ⟨1⟩.

So far, we have shown that for the sequence (A), (B), (D), at node (B), we could perform the following kinds of deletions: (1) delete nothing, (2) delete φ alone, (3) delete both of $Y \leq 0$ and $Y \geq 0$, and (4) delete all of $\varphi, Y \leq 0$ and $Y \geq 0$. (That is, we exclude the case where we delete just one of $Y \leq 0$ and $Y \geq 0$.) We would then have a new sequence where (B) continues to subsume (D).

Let us now examine the sequence (A), (B), (E), which is the second derivation sequence emanating from (B). Note that (E) is a target goal, and we require that $Y \geq 0$ here. Thus the choices (3) or (4) above made for the sequence (A), (B), (D) would not be suitable for this path, because these deletions would remove all constraints on Y.

It thus becomes clear that the best choice is (2). That is, we end up generalizing (B) by deleting only the constraint φ. With this as an interpolant, it is now no longer necessary to traverse the subtree of goal (C).

In summary, a final goal that is subsumed by one of its ancestors is already labelled with a *path invariant*, that is, a invariant that holds only for the particular derivation sequence. However, it is still possible to generalize the final goal by deleting any individual constraint that also appears at the ancestor goal, and it is *itself invariant*. Note that in this example, φ was invariant through the cycle, unlike $Y \leq 0$ or $Y \geq 0$. A rather straightforward idea then is to consider only invariant constraints as candidates for deletion within a sequence. We detail this in sections 4 and 5.2.

3 CLP Preliminaries

We first briefly overview CLP [29]. The *universe of discourse* is a set of terms, integers, and arrays of integers. A *constraint* is written using a language of functions and relations. In this paper, we will not define the constraint language explicitly, but invent them on demand in accordance with our examples. Thus the terms of our CLP programs include the function symbols of the constraint language.

An *atom*, is as usual, of the form $p(\tilde{t})$ where p is a user-defined predicate symbol and the $\tilde{t}$ a tuple of terms. A *rule* is of the form $A\texttt{:-}\tilde{B}, \phi$ where the atom A is the *head* of the rule, and the sequence of atoms $\tilde{B}$ and the constraint ϕ constitute the *body* of the rule. A *goal* G has exactly the same format as the body of a rule. We say that a rule is a (constrained) *fact* if $\tilde{B}$ is the empty sequence. A *ground instance* of a constraint, atom and rule is defined in the obvious way.

A *substitution* simultaneously replaces each variable in a term or constraint into some expression. We specify a substitution by the notation $[\tilde{E}/\tilde{X}]$, where $\tilde{X}$ is a sequence $X_1, \ldots, X_n$ of variables and $\tilde{E}$ a list $E_1, \ldots, E_n$ of expressions, such that X_i is replaced by E_i for all $1 \leq i \leq n$. Given a substitution θ, we write as $E\theta$ the application of the substitution to an expression E. A *renaming* is a substitution which maps variables variables. A *grounding* is a substitution which maps each variable into a value in its domain.

In this paper we deal with goals of the form $p_k(\tilde{X}), \Psi(\tilde{X})$, where p_k is the predicate defined in a CLP model of an imperative program and $\Psi(\tilde{X})$ is a constraint on $\tilde{X}$. Given a goal $\mathcal{G} \equiv p_k\tilde{X}), \Psi(\tilde{X})$, $[\![\mathcal{G}]\!]$ is the set of the groundings θ of the primary variables $\tilde{X}$ such that $\tilde{\exists}\Psi(\tilde{X})\theta$ holds. We say that a goal $\overline{\mathcal{G}} \equiv p_k(\tilde{X}), \overline{\Psi}(\tilde{X})$ *subsumes* another goal $\mathcal{G} \equiv p_{k'}(\tilde{X}'), \Psi(\tilde{X}')$ if $k = k'$ and $[\![\overline{\mathcal{G}}]\!] \supseteq [\![\mathcal{G}]\!]$. Equivalently, we say that $\overline{\mathcal{G}}$ is a *generalization* of $\mathcal{G}$. We write $\mathcal{G}_1 \equiv \mathcal{G}_2$ if $\mathcal{G}_1$ and $\mathcal{G}_2$ are generalizations of each other. We say that a sequence is *subsumed* if its last goal is subsumed by another goal in the sequence.

Given two goals $\mathcal{G}_1 \equiv p_k(\tilde{X}_1), \Psi_1$ and $\mathcal{G}_2 \equiv p_k(\tilde{X}_2), \Psi_2$ sharing a common program point k, and having disjoint sets of variables, we write $\mathcal{G}_1 \wedge \mathcal{G}_2$ to denote the goal $p_k(\tilde{X}_1), (\tilde{X}_1 = \tilde{X}_2, \Psi_1, \Psi_2)$.

Let $G \equiv (B_1, \cdots, B_n, \phi)$ and P denote a goal and program respectively. Let $R \equiv A\texttt{:-}C_1, \cdots, C_m, \phi_1$ denote a rule in P, written so that none of its variables appear in G. Let $A = B$, where A and B are atoms, be shorthand for equations between their corresponding arguments. A *reduct* of G using R is of the form

$(B_1, \cdots, B_{i-1}, C_1, \cdots, C_m, B_{i+1}, \cdots, B_n, B_i = A \wedge \phi \wedge \phi_1)$

provided $B_i = A \wedge \phi \wedge \phi_1$ is satisfiable.

A *derivation sequence* is a possibly infinite sequence of goals $G_0, G_1, \cdots$ where $G_i, i > 0$ is a reduct of G_{i-1}. If there is a last goal G_n with no atoms called *terminal goal*, we say that the derivation is *successful*. In order to prove safety, we test that the goal implies the safety condition. A derivation is ground if every reduction therein is ground. Given a sequence τ defined to be $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n$, then *cons*$(\tau)$ is all the constraints of the goal $\mathcal{G}_n$. We say that a sequence is *feasible* if *cons*(τ) is satisfiable, and *infeasible* otherwise. Moreover, we say that a derivation sequence τ is *successful*, when it is feasible and k is the final point.

The *derivation tree* of a CLP has as branches its derivation sequences. In this tree, the *ancestor-descendant* relation between nodes is defined in the usual way. A leaf of this tree is *cyclic* if its program point appears at one of its ancestors, which will be called the *cyclic ancestor* of the leaf in question. A derivation tree is *closed* if all its branches are either successful, infeasible, or subsumed. Given a CLP with a derivation tree T, whose root is a goal $\mathcal{G}$, we denote by $T[\mathcal{G}'/\mathcal{G}]$ the tree obtained by replacing the root $\mathcal{G}$ by a new goal $\mathcal{G}'$, and relabeling the nodes of T to reflect the rules represented by the edges of T. In other words, $T[\mathcal{G}'/\mathcal{G}]$ represents the symbolic computation tree of the same program, started at a different goal.

Informally, we say that two closed trees T and T' have the *same shape* if their sequences can be uniquely paired up such that, for every pair of sequences (τ, τ'), we have: (a) τ is a sequence in T, and τ' is a sequence in T'; (b) τ and τ' have the same sequence of predicates; and (c) τ and τ' are both simultaneously either successful, infeasible, or subsumed.

Given a target property represented as a condition $\Psi(\tilde{X})$ on system variables, we say that a final goal $\mathcal{G}$ is *safe* if $[\![\mathcal{G}]\!] \subseteq [\![p_k(\tilde{X}), \Psi(\tilde{X})]\!]$.

We end this section with a definition of the notion of interpolant.

Definition 1 (Interpolant). *A goal $\mathcal{G}_I$ is an* interpolant *for closed tree T with root $\mathcal{G}$ if:*
- *all its successful sequences end in safe goals;*
- *$\mathcal{G}_I$ subsumes $\mathcal{G}$,*
- *T and $T[\mathcal{G}_I/\mathcal{G}]$ have the same shape.* □

4 The Algorithm

In this section, we describe an idealized algorithm to traverse a computation tree of a given goal $\mathcal{G}$. The recursive procedure computes for each such $\mathcal{G}$ a possibly infinite set of interpolants. These interpolants are then propagated to the parent goal $\mathcal{G}_p$ of $\mathcal{G}$. The eventual completion of the traversal of $\mathcal{G}_p$ is likewise augmented by a computation of its interpolant. Clearly this process is most naturally implemented recursively by a depth-first traversal. Our main technical result is that all interpolants are *safe*, that is, all computation trees resulting from a goal subsumed by an interpolant are safe.

The algorithm is presented in Fig. 6. Its input is a goal $\mathcal{G}$. We assume that there is a target property that $\Psi_f(\tilde{X}_f)$ must hold at a target point k_f. Without loss of generality, we also assume that $\mathcal{G}$ and Ψ_f do not share variables, and before the execution, the memo table containing computed interpolants is empty. The idea is to compute interpolants that are as general as possible. The function `solve` is initially called with the initial goal. We first explain several subprocedures that are used in Fig. 6:

- `memoed(`$\mathcal{G}$`)` tests if $\mathcal{G}'$ is in the memo table such that $\mathcal{G}'$ subsumes $\mathcal{G}$. If this is the case, it returns the set of all such $\mathcal{G}'$.

- `memo(`$\mathcal{I}$`)` records the set $\mathcal{I}$ of interpolants in the memo table.
- $\text{WP}(\mathcal{G}, \rho)$ is a shorthand for the condition $\rho \models \Psi'$ where ρ is the constraint in the rule used to produce the reduct $\mathcal{G} \equiv p_k(\tilde{X}'), \Psi'$; $\tilde{X}$ and $\tilde{X}'$ are the variables appearing in this rule.

In what follows, we discuss each of the five cases of our algorithm. Each case is characterized by a proposition that contributes to the proof of the correctness theorem stated at the end of this section.

First, the algorithm tests whether the input goal $\mathcal{G}$ is already memoed, in which case, the return value of `memoed(`$\mathcal{G}$`)` is returned. The following proposition holds:

Proposition 1. *If $\mathcal{G}' \in$ `memoed`$(\mathcal{G})$ then $\mathcal{G}'$ is an interpolant of $\mathcal{G}$.*

Next consider the case where current goal $\mathcal{G}$ is *false*. Here the algorithm simply returns a singleton set containing the goal $p_k(\tilde{X}), \mathit{false}$. The following proposition is relevant to the correctness of this action.

Proposition 2. *The goal $p_k(\tilde{X}), \mathit{false}$ is an interpolant of a false goal.*

Next consider the case where current goal $\mathcal{G}$ is terminal. If $\mathcal{G}$ is unsafe, that is $[\![\mathcal{G}]\!] \not\subseteq [\![p_k(\tilde{X}_f), \Psi_f(\tilde{X}_f)]\!]$, the entire function aborts, and we are done. Otherwise, the algorithm returns *all* generalizations $\overline{\mathcal{G}}$ of $\mathcal{G}$ such that $[\![\overline{\mathcal{G}}]\!] \subseteq [\![p_k(\tilde{X}_f), \Psi_f(\tilde{X}_f)]\!]$. The following proposition is relevant to the correctness of this action.

Proposition 3. *When the target property is specified by $p_{k_f}(\tilde{X}_f), \Psi_f$ where k_f is a final program point, and the goal $\mathcal{G}$ is safe, then its generalization $\overline{\mathcal{G}}$ is an interpolant of $\mathcal{G}$, where $[\![\overline{\mathcal{G}}]\!] \subseteq [\![p_{k_f}(\tilde{X}_f), \Psi_f]\!]$.*

Next consider the case where current goal $\mathcal{G}$ is a looping goal, that is, $\mathcal{G}$ is subsumed by an ancestor goal. Here we compute a set of generalizations of the ancestor goal such that the same execution from the ancestor goal to the current input goal still results in a cycle. In other words, we return the set of all possible generalizations of the ancestor goal such that when the same reduction sequence (with constraint Φ along the sequence) is traversed to the current goal, the goal remains subsumed by the ancestor goal. The following proposition is relevant to the correctness of this action.

Proposition 4. *If $\mathcal{G} \equiv p_k(\tilde{X}), \Psi$ is a goal with ancestor $p_k(\tilde{X}'), \Psi'$ such that $\Psi \equiv \Psi' \wedge \Phi$, then $p_k(\tilde{X}), \overline{\Psi}[\tilde{X}/\tilde{X}']$ is an interpolant of $\mathcal{G}$ where $\overline{\Psi} \wedge \Phi \models \overline{\Psi}[\tilde{X}/\tilde{X}']$ if all successful goals in the tree are safe.*

Finally we consider the recursive case. The algorithm represented by the recursive procedure `solve`, given in Figure 6, applies all applicable CLP rules to create new goals from $\mathcal{G}$. It does this by reducing $\mathcal{G}$. It then performs recursive calls using the reducts. Given the return values of the recursive calls, the algorithm computes the interpolants of the goal $\mathcal{G}$ by an operation akin to weakest precondition propagation. The final set of interpolants for $\mathcal{G}$ is then simply the intersection of the sets of interpolants returned by recursive calls.

Proposition 5. *Let $\mathcal{G}$ have the reducts $\mathcal{G}_i$ where $1 \leq i \leq n$. Let $\overline{\mathcal{G}_i} \in$ `solve`$(\mathcal{G}_i)$, $1 \leq i \leq n$. Then $\overline{\mathcal{G}}$ is an interpolant of $\mathcal{G}$ if for all $1 \leq i \leq n$, $\overline{\mathcal{G}} \equiv \cap_{i=1}^{n} \{p_k(\tilde{X}), \text{WP}(\overline{\mathcal{G}_i}, \rho_i)\}$.*

```
solve($\mathcal{G} \equiv p_k(\tilde{X}), \Psi$) returns  a set of interpolants
    case ($\mathcal{I}$ = memoed($\mathcal{G}$)): return $\mathcal{I}$
    case $\mathcal{G}$ is false ($\Psi \equiv \mathit{false}$): return $\{p_k(\tilde{X}), \mathit{false}\}$
    case $\mathcal{G}$ is target ($k = k_f$):
        if ($\Psi[\tilde{X}_f/\tilde{X}] \not\models \Psi_f$) abort else return $\{\overline{\mathcal{G}} : [\![\overline{\mathcal{G}}]\!] \subseteq [\![p_{k_f}(\tilde{X}_f), \Psi_f]\!]\}$
    case $\mathcal{G}$ is cyclic:
        let cyclic ancestor of $\mathcal{G}$ be $p_k(\tilde{X}'), \Psi'$ and $\Psi \equiv \Psi' \wedge \Phi$
        return $\{p_k(\tilde{X}), \overline{\Psi}[\tilde{X}/\tilde{X}'] : \overline{\Psi} \wedge \Phi \models \overline{\Psi}[\tilde{X}/\tilde{X}']\}$
    case otherwise:
        foreach rule $p_k(\tilde{X})$ :- $p_{k'}(\tilde{X}'), \rho(\tilde{X}, \tilde{X}')$:
            $\mathcal{I}$ := $\mathcal{I}$ $\cap$ $\{p_k(\tilde{X}), \textsc{wp}(\overline{\mathcal{G}}', \rho) : \overline{\mathcal{G}}' \in$ solve($p_{k'}(\tilde{X}'), \Psi \wedge \rho$)$\}$
        endfor
        memo($\mathcal{I}$) and return $\mathcal{I}$
```

Fig. 6. The Interpolation Algorithm

The following theorem follows from Propositions 1 through 5:

Theorem 1 (Safety). *The algorithm in Fig. 6 correctly returns interpolants.*

We note that the generation of interpolants here employs a notion of *weakest precondition* in the literature [28, 30]. Given a goal transition induced by a statement s and represented as input-output constraint $\rho(\tilde{X}, \tilde{X}')$, the weakest precondition of a condition Ψ' is $\rho(\tilde{X}, \tilde{X}') \models \Psi'$. By our use of interpolation, we do not directly use the weakest precondition to generalize a goal, a technique which is notoriously inefficient [31], but we instead use interpolants, which approximate the weakest precondition, are efficient to compute, and yet still generalize the input goal. That is, instead of WP, we use another function $\textsc{intp}(\mathcal{G}, \rho)$ such that when $\mathcal{G}$ is $p_{k'}(\tilde{X}'), \Psi'$, then $\textsc{intp}(\mathcal{G}, \rho)$ is a constraint $\overline{\Psi}$ such that $\overline{\Psi}$ entails $(\rho \models \Psi')$. In the next section, we demonstrate an algorithm that implements INTP based upon an efficient implementation of constraint deletions and "slackening."

5 Serial Constraint Replacement

We now present a general practical approach for computing an interpolant. Recall the major challenges arising from the idealized algorithm in Fig. 6:

- not one but a *set* of interpolants is computed for each goal traversed;
- even for a single interpolant, there needs to be efficient way to compute it;
- interpolants for the descendants of a goal need to be combined by a process akin to weakest-precondition propagation, and then these results need to be conjoined.

Given a set of constraints, we would like to generalize the maximal number of constraints that would preserve some "interpolation condition". Recall that this condition is (a) being unsatisfiable in the case we are dealing with a terminal (*false*) goal, (b) implying a target property in case we are dealing with a target goal, or (c) implying that the subsumed terminal node remains subsumed.

Choosing a subset of constraints is clearly an inefficient process because there are an exponential number of subsets to consider. Instead, we order the constraints according to the execution order, and process the constraints *serially*. While not guaranteed to find the smallest subset, this process is efficient and more importantly, attempts to generalize the constraints *in the right order* because the earliest constraints, those that appear in the most goals along a path, are generalized first.

The computation of interpolants is different across the three kinds of paths considered. Case (a) and (b) are similar and will be discussed together, and we discuss separately case (c).

5.1 Sequences ending in a False or Target Goal

Consider each constraint Ψ along the path to the terminal goal in turn. If the terminal goal were *false*, we replace Ψ with a more general constraint if the goal remains *false*, In case the terminal goal were a target, we replace Ψ with a more general constraint if the goal remains safe. We end up concretely with a subset of the constraints in the terminal goal, and this defines that the interpolant is the goal with the replacements realized. We next exemplify.

Note that a program statement gives rise to a constraint relating the states before and after execution of the statement Consider the following imperative program and its CLP model:

```
⟨0⟩ x := x+1              p0(X,Y,Z) :- p1(X',Y,Z),X' = X+1.
⟨1⟩ if (z ≥ 0) then       p1(X,Y,Z) :- p2(X,Y,Z),Z ≥ 0.
⟨2⟩     y := x            p1(X,Y,Z) :- p3(X,Y,Z),Z < 0.
⟨3⟩                       p2(X,Y,Z) :- p3(X,Y',Z),Y' = X.
                          p3(X,Y,Z).
```

The sequence of constraints obtained from the derivation sequence which starts from the goal $p_0(X_0,Y_0,Z_0),X_0 = 0,Y_0 = 2$ and goes along $\langle 0\rangle$, $\langle 1\rangle$, $\langle 2\rangle$, to $\langle 3\rangle$ is $X_0 = 0,Y_0 = 2,X_1 = X_0+1,Z_0 \geq 0,Y_1 = X_1$ for some indices 0 and 1 denoting versions of variables. At this point, we need to *project* the constraints onto the *current variables*, those that represent the current values of the original program variables. At $\langle 3\rangle$, the projection of interest is $X = 1,Y = 1,Z \geq 0$.

If the target property were $Y \geq 0$ at point $\langle 3\rangle$, then it holds because the projection implies it. In the case of an infeasible sequence (not the case here), our objective would be to preserve the infeasibility, which means that we test that the constraints imply the target condition *false*.

In general, then, we seek to generalize a projection of a list of constraints in order to obtain an interpolant. Here, we simply replace a constraint with a more general one as long as the result satisfies the target property. For the example above, we could delete (replace with *true*) the constraints $Y_0 = 2$ and $Z_0 \geq 0$ and still prove the target property $Y \geq 0$ at $\langle 3\rangle$.

In Table 1 we exemplify both the deletion and the slackening techniques using our running example. The first column of Table 1 is the executed statements column (we represent the initial goal in curly braces). During the first traversal without abstraction,

Statement	No Interpolation		Deletion		Deletion and Slackening (1)	
	Constraint	Projection	Constraint	Projection	Constraint	Projection
$\{x = 0, y = 2\}$ $\langle 0\rangle$	$X_0 = 0, Y_0 = 2$	$X = 0, Y = 2$	$X_0 = 0$	$X = 0$	$X_0 \geq 0$	$X \geq 0$
$x := x+1$ $\langle 1\rangle$	$X_1 = X_0 + 1$	$X = 1, Y = 2$	$X_1 = X_0 + 1$	$X = 1$	$X_1 = X_0 + 1$	$X \geq 1$
if $(z \geq 0)$ $\langle 2\rangle$	$Z_0 \geq 0$	$X = 1, Y = 2, Z \geq 0$	(none)	$X = 1$	(none)	$X \geq 1$
$y := x$ $\langle 3\rangle$	$Y_1 = X_1$	$X = 1, Y = 1, Z \geq 0$	$Y_1 = X_1$	$X = 1, Y = 1$	$Y_1 = X_1$	$X \geq 1, Y = X$

Table 1. Interpolation Techniques

the constraints in the second column is accumulated. The projection of the accumulated constraints into the primary variables is shown in the third column. As mentioned, this execution path satisfies the target property $Y_1 \geq 0$. We generalize the goals along the path using one of two techniques:

- **Constraint deletion.** Here we replace a constraint with *true*, effectively deleting it. This is demonstrated in the fourth column of Table 1. Since the constraint $Z_0 \geq 0$ and $Y_0 = 2$ do not affect the satisfaction of the target property, they can be deleted. The resulting projections onto the original variables is shown in the fifth column, effectively leaving Y unconstrained up to $\langle 2\rangle$, while removing all constraints on Z.
- **Slackening.** Another replacement techniqe is by replacing equalities with non-strict inequalities, which we call *slackening*. For example, replacing the constraint $X_0 = 0$ with $X_0 \geq 0$ in the fifth column of Table 1 would not alter the unsatisfiability of the constraint system. (We would repeat this exercise with $X_0 \leq 0$.) The actual replacement in the sixth column results in the more general interpolants in the seventh column. Recall the demonstration of slackening in Section 2.

5.2 Sequences ending in a Subsumed Goal

Consider now the case of a sequence τ ending in a goal which is subsumed by an ancestor goal. Say τ is $\tau_1\tau_2$ where τ_1 depicts the prefix sequence up to the subsuming ancestor goal. The subsumption property can be expressed as

$$cons(\tau) \models cons(\tau_1)[\tilde{X}_i/\tilde{X}]$$

Following the spirit of the previous subsection, we now seek to replace any individual constraint in τ_1 as long as this subsumption holds. (Note that replacing a constraint in τ_1 automatically replaces a constraint in τ, because τ_1 is a prefix of τ.)

However, there is one crucial difference with the previous subsection. Here we shall only be replacing an individual constraint Ψ that is *itself invariant* (for point k) in the sequence. The reason for this is based on the fact that in order to propagate an interpolant (now represented as a single goal, and not a family), the interpolants for descendant nodes need to be simply conjoined in order to form the interpolant for the parent goal (explained in the next section). This may result in re-introduction of a replaced constraint Ψ in τ_1. The condition that Ψ is itself invariant guarantees that even if Ψ is re-introduced, we still have an interpolant that is invariant for the cycle.

5.3 Propagating Interpolants

One key property of serial constraint replacement is the ease with which interpolants are generated from various derivation paths that share some prefix. Recall in the previous sections that we need to produce a common interpolant for the intermediate nodes in the tree from the interpolants of their children. Here, we compute candidate interpolants of a parent node from the interpolants of the children. Note that each interpolant is now simply a conjunction of constraints. Then, the interpolant of the parent is *simply the conjunction* of the candidate interpolants (cf. the intersection of interpolants for the recursive case of the algorithm in Section 4).

Statement	Deletion and Slackening (2)		Combination (1), (2)	
	Constraint	Projection	Constraint	Projection
$\{x = 0, y = 2\}$ $\langle 0 \rangle$	$Y_0 \geq 1$	$Y \geq 1$	$X_0 \geq 0, Y_0 \geq 1$	$X \geq 0, Y \geq 1$
$x := x + 1$ $\langle 1 \rangle$	(none)	$Y \geq 1$	$X_1 = X_0 + 1$	$X \geq 1, Y \geq 1$
if $(z \geq 0)$ $\langle 3 \rangle$	(none)	$Y \geq 1$		

Table 2. Propagating Interpolants

Let us now consider another path through the sample program, one that visits $\langle 0 \rangle$, $\langle 1 \rangle$, and $\langle 3 \rangle$ without visiting $\langle 2 \rangle$. The statements and the interpolation along this path by deletion and slackening are shown in the first to third columns of Table 2. Note that this path, and the path $\langle 0 \rangle$, $\langle 1 \rangle$, $\langle 2 \rangle$, and $\langle 3 \rangle$ considered before, share the initial goal $p_0(X_0, Y_0, Z_0), X_0 = 0, Y_0 = 2$ and the first statement of the program. Now to compute the actual interpolant for $\langle 0 \rangle$ and $\langle 1 \rangle$, we need to consider the interpolants generated from both paths. Using our technique, we can simply conjoin the constraints at each level to obtain the common interpolants. This is exemplified in the fourth column of Table 2. As can be seen, the resulting interpolants are simple because they do not contain disjunction. The resulting projection is shown in the fifth column. The execution of the program, starting from the goal represented by the projection, along either of the possible paths, is guaranteed to satisfy the target property.

6 Experimental Evaluation

We implemented a prototype verifier using CLP$(\mathcal{R})$ constraint logic programming system [32]. This allows us to take advantage of built-in Fourier-Motzkin algorithm [33] and meta-level facilities for the manipulation of constraints. We performed the experiments on a 1.83GHz Macbook Pro system.

6.1 Array Bounds Verification by Constraint Deletion

We verify that at each array access point, the possible index values are legal. The programs "FFT" and "LU" are from the Scimark benchmark suite, and "linpack" from Toy [34]. Here we manually provide an invariant for each loop to finitize the traversal. The specific interpolation method used is the constraint deletion.

In Table 3, "Goals" indicates the cost of traversal, and the time is in seconds. The fairly large "linpack" program (907 LOC) is run in four cases.

Problem	LOC	No Interpolation		Interpolation	
		States	Time	States	Time
FFT	165	2755	10.62	120	0.10
LU	102	298	0.39	138	0.17
linpack^{200}	907	6385	19.70	332	0.53
linpack^{400}	907	8995	151.65	867	2.47
linpack^{600}	907	16126	773.63	867	2.47
linpack	907	∞	∞	867	2.47

Table 3. Array Bounds Verification

For the first three, we apply a depth bound for the search tree, progressively at 200, 400, and 600 (denoted in the table as linpack^b, where b is the depth bound) to demonstrate the change in the size of the search tree, which is practically infinite when there is no depth bound. Using interpolation, on the other hand, we can completely explore the computation tree. In all cases, the number of goals visited is significantly reduced as a result of interpolation.

6.2 Resource Bound Verification by Constraint Deletion and Slackening

Here we seek to verify an upper bound for a certain variable, representing resource usage. In our examples, the resource of interest is the execution time of a program, modeled as a monotonically increasing instrumented variable.

Problem	LOC	No Interpolation		Interpolation	
		States	Time	States	Time
decoder	27	344	1.42	160	0.49
sqrt	16	923	27.13	253	7.46
qurt	40	1104	38.65	290	11.39
janne_complex	15	1410	48.64	439	7.87
statemate^{20}	1298	21	0.05	21	0.08
statemate^{30}	“	1581	2.93	48	0.16
statemate^{40}	“	∞	∞	71	0.24
statemate	“	∞	∞	1240	17.09

Table 4. Resource Bound Verification

We consider the “decoder”, “sqrt”, “qurt”, “janne_complex” and “statemate” programs from the Mälardalen benchmark [35] used for testing WCET (worst-case execution time) analysis. One challenge to accurate WCET is that program fragments can behave significantly different when invoked in different contexts. For the “statemate” problem, we limit the depth bound of the goal-space search without interpolation into cases 20, 30, and 40, and we also store the summarization of the maximum resource usage of a computation subtree in the memo table. In Table 4, “statemate^n” denotes verification runs of “statemate” with n as the the depth bound. For “statemate,” we limit the number of iteration of a loop in the program to two (actual number of iterations when all variables are initialized to 0). The “statemate” program displays a significant amount of dependencies between Boolean conditions in a path. The more dependency between statements there is, the less the reduction that can be obtained by interpolation. For example, in the experiment “statemate^{30}” the reduction in goal space from 1581 to 48. More notably, the interpolation based goal exploration can in fact completely explore the goal space for the “statemate” experiment, traversing just 1240 goals.

Finally, we compare with the CEGAR tool BLAST, on a fragment [2], of “statemate.” As in BLAST, we combine statements in a block into single transition to facilitate proper comparison of the number of search tree nodes. Our prototype completed the verification in traversing 142 search tree nodes. With default options (breadth-first, no heuristics), BLAST traverses 1410 nodes. The difference is essentially due to spurious paths and constraint slackening.

[2] BLAST ran out of memory when run with the full program.

Acknowledgement

We thank Dirk Beyer for his help on BLAST.

References

1. Marriott, K., Stuckey, P.J.: Programming with Constraints. MIT Press (1998)
2. Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. J. ACM **43**(1) (January 1996) 20–74
3. Swift, T.: A new formulation of tabled resolution with delay. In Barahona, P., Alferes, J.J., eds.: 9th EPIA. Volume 1695 of LNCS., Springer (1999) 163–177
4. Sagonas, K., Swift, T., Warren, D.S., Freire, J., Rao, P., Cui, B., Johnson, E., de Castro, L., Dawson, S., Kifer, M.: The XSB System Version 2.5 Volume 1: Programmer's Manual. (June 2003)
5. Jaffar, J., Santosa, A.E., Voicu, R.: Efficient memoization for dynamic programming with ad-hoc constraints. In: 23rd AAAI, AAAI Press (2008) 297–303
6. Frost, D., Dechter, R.: Dead-end driven learning. In: 12th AAAI, AAAI Press (1994) 294–300
7. Bayardo, Jr., R.J., Schrag, R.: Using csp look-back techniques to solve real-world sat instances. In: 14th AAAI/9th IAAI, AAAI Press (1997) 203–208
8. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In Cleaveland, R., ed.: 5th TACAS. Volume 1579 of LNCS., Springer (1999) 193–207
9. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: 38th DAC, ACM Press (2001) 530–535
10. Silva, J.P.M., Sakallah, K.A.: GRASP—a new search algorithm for satisfiability. In: ICCAD 1996, ACM and IEEE Computer Society (1996) 220–227
11. McMillan, K.L.: Applying SAT methods in unbounded symbolic model checking. Lecture Notes in Computer Science **2404** (2002) 250–??
12. Lahiri, S.K., Nieuwenhuis, R., Oliveras, A.: SMT techniques for fast predicate abstraction. [36] 424–437
13. Jones, R.B., Dill, D.L., Burch, J.R.: Efficient validity checking for processor verification. In Rudell, R.L., ed.: ICCAD 1995, IEEE Computer Society Press (1995) 2–6
14. Barrett, C., Dill, D.L., Levitt, J.R.: Validity checking for combinations of theories with equality. In Srivas, M.K., Camilleri, A.J., eds.: 1st FMCAD. Volume 1166 of LNCS., Springer (1996) 187–201
15. Stump, A., Barrett, C., Dill, D.L.: CVC: A cooperating validity checker. In Brinksma, E., Larsen, K.G., eds.: 14th CAV. Volume 2404 of LNCS., Springer (2002) 500–504
16. Barrett, C., Berezin, S.: CVC Lite: A new implementation of the cooperating validity c hecker. In Alur, R., Peled, D.A., eds.: 16th CAV. Volume 3114 of LNCS., Springer (2004)
17. McMillan, K.L.: Interpolation and SAT-based model checking. In: 15th CAV. Volume 2725 of LNCS., Springer (2003) 1–13
18. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient interpolant generation in satisfiability modulo theo ries. In Ramakrishnan, C.R., Rehof, J., eds.: 14th TACAS. Volume 4963 of LNCS., Springer (2008) 397–412
19. McMillan, K.L.: An interpolating theorem prover. TCS **345**(1) (2005) 101–121
20. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5) (September 2003) 752–794
21. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: 29th POPL, ACM Press (2002) 58–70 SIGPLAN Notices 37(1).

22. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: 31st POPL, ACM Press (2004) 232–244
23. McMillan, K.L.: Lazy abstraction with interpolants. [36] 123–136
24. Beyer, D., Henzinger, T., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. STTT **9** (2007) 505–525
25. Flanagan, C.: Automatic software model checking using CLP. In Degano, P., ed.: 12th ESOP. Volume 2618 of LNCS., Springer (2003) 189–203
26. Jaffar, J., Santosa, A.E., Voicu, R.: Modeling systems in CLP. In Gabbrielli, M., Gupta, G., eds.: 21st ICLP. Volume 3668 of LNCS., Springer (2005) 412–413
27. Delzanno, G., Podelski, A.: Constraint-based deductive model checking. Int. J. STTT **3**(3) (2001) 250–270
28. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall Series in Automatic Computation. Prentice-Hall (1976)
29. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. J. LP **19/20** (May/July 1994) 503–581
30. Bjørner, N., Browne, A., Manna, Z.: Automatic generation of invariants and intermediate assertions. TCS **173**(1) (February 1997) 49–87
31. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: Generating compact verificatio n conditions. In: 28th POPL, ACM Press (2001) 193–205
32. Jaffar, J., Michaylov, S., Stuckey, P.J., Yap, R.H.C.: The CLP($\mathcal{R}$) language and system. ACM TOPLAS **14**(3) (1992) 339–395
33. Jaffar, J., Maher, M.J., Stuckey, P.J., Yap, R.H.C.: Output in CLP($\mathcal{R}$). In: Proc. Int. Conf. on Fifth Generation Computer Systems, Tokyo, Japan. Volume 2. (1992) 987–995
34. Toy, B.: Linpack.c (1988) URL http://www.netlib.org/benchmark/linpackc.
35. : Mälardalen WCET research group benchmarks. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html` (2006)
36. Ball, T., Jones, R.B., eds.: Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings. In Ball, T., Jones, R.B., eds.: 18th CAV. Volume 4144 of LNCS., Springer (2006)