Failure Tabled Constraint Logic Programming by Interpolation ∗

Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey
Department of Computing and Information Systems
The University of Melbourne, Victoria 3010, Australia
(*e-mail:* {gkgange,jorge.navas,schachte,harald,pstuckey}@unimelb.edu.au)

Abstract

We present a new execution strategy for constraint logic programs called *Failure Tabled CLP*. Similarly to *Tabled CLP* our strategy records certain derivations in order to prune further derivations. However, our method only learns from *failed derivations*. This allows us to compute *interpolants* rather than *constraint projection* for generation of *reuse conditions*. As a result, our technique can be used where projection is too expensive or does not exist. Our experiments indicate that Failure Tabling can speed up the execution of programs with many redundant failed derivations as well as achieve termination in the presence of infinite executions.

1 Introduction

Constraint Logic Programming (CLP) (Jaffar and Lassez 1987) has been successfully used in many different contexts such as management decision problems, trading, scheduling, electrical circuit analysis, mapping in genetics, *etc.* (Marriott and Stuckey 1998). However, its standard execution model based on *depth-first search* with *left-to-right clause selection* suffers from two major drawbacks inherited from *Logic Programming* (LP):

1. the derivation tree containing all program executions can be huge even if many subtrees may be redundant, and
2. it is incomplete in the presence of infinite derivations since the execution may choose these rather than executing the finite ones.

To tackle these issues, an alternative LP execution strategy called *Tabling* was proposed (Tamaki and Sato 1986; Warren 1992). This strategy records calls and their answers, for reuse in future calls. *Tabled Constraint Logic Programming* (TCLP) (Codognet 1995) is a natural extension of Tabling to CLP programs. TCLP makes explicit the requirement of the tabling execution on the constraint domain. For instance, to detect when a more particular call can consume answers from a more general one, it uses *constraint entailment*. And for determining the calling constraint for a tabled call, it needs to make use of *constraint projection*. The projection operation is a particularly onerous requirement. Many constraint domains have no projection operation (or weak projection only) (Marriott and Stuckey 1998), and for those that do, the cost is often prohibitive.

∗ We wish to thank Jose. F. Morales for providing support integrating MathSAT into Ciao and Manuel Carro and Corneliu Popeea for fruitful discussions about tabling and interpolation, respectively. We acknowledge support of the Australian Research Council through Discovery Project Grant DP110102579.

```
?-  0 ≤ X ≤ 5, 0 ≤ Y ≤ 3, R ≥ 15, p1(X, Y, R) .

p1(X1, Y1, R1) :- X1' = X1 + Y1 + 2, p2(X1', Y1, R1) .
p1(X2, Y2, R2) :- X2' = X2 + Y2 + 1, /*?*/ p2(X2', Y2, R2) .

p2(X3, Y3, R3) :- Y3' = Y3 + 1, p3(X3, Y3', R3) .
p2(X4, Y4, R4) :- Y4' = Y4 + 2, p3(X4, Y4', R4) .

p3(X5, Y5, R5) :- R5 = X5 + Y5 − 1 .
p3(X6, Y6, R6) :- R6 = X6 + Y6 .
```

Fig. 1. A recursion-free CLP program with redundant derivations

We present *Failure Tabled Constraint Logic Programming* (FTCLP), a new execution strategy that augments the classical top-down CLP execution algorithm to benefit from pruning redundant failed derivations that can avoid non-terminating executions without the use of constraint projection. Our algorithm executes the CLP program in a top-down manner while labelling nodes in the derivation tree with two kinds of objects:

1. A *reuse condition* is a formula that, if it is implied by a goal's *constraint store* (the constraints accumulated during the execution of the derivation of the goal), then it is guaranteed that no new answers can be generated, and thus, the search can stop and backtrack to another choice point.
2. A *set of answers* for a goal is the constraint stores resulting from the successful derivations for the goal.

Whenever a new goal is executed and its constraint store implies the reuse condition of a recorded goal, the current CLP execution can be stopped and the set of answers from the more general state can be consumed without the need of running that goal. We show that this is possible even in the presence of recursive clauses with infinite derivations. To generate reuse conditions we use interpolation (Craig 1957), a technique that has attracted much interest in counterexample-driven verification during the last decade.

We start by describing our method through a recursion-free CLP program. Later, we will show how to deal with recursion and infinite derivations.

Example 1 (*Recursion-Free Clauses*)
Consider the query/program in Figure 1. 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 successful derivation is marked with a (green) "tick" symbol and a failed derivation with a (red) "cross" symbol. It is easy to check that this tree has seven failed derivations in addition to the successful one yielding $(X = 5, Y = 3, R = 15)$. Our method takes advantage of the fact that some failed derivations can be summarized by compact explanations which can be used to produce a smaller derivation tree while preserving all original answers. The non-dashed fragment of Figure 2 gives the derivation tree computed by our method. Note that the clauses of `p2/3` are explored only once.

From the leftmost derivation we collect in π all its constraints (including both primitive and user-defined ones). That is, $\pi \equiv [0 \leq X \leq 5, 0 \leq Y \leq 3, R \geq 15,$ `p1(`X, Y, R`)`, $X_1' = X + Y + 2,$ `p2(`X_1', Y, R`)`, $Y_3' = Y + 1,$ `p3(`X_1', Y_3', R`)`, $R = X_1' + Y_3' - 1]$. Note that we rename variables accordingly whenever there is a match between a body atom and a clause head. For instance, the match between the atom `p1(`X, Y, R`)` and the head

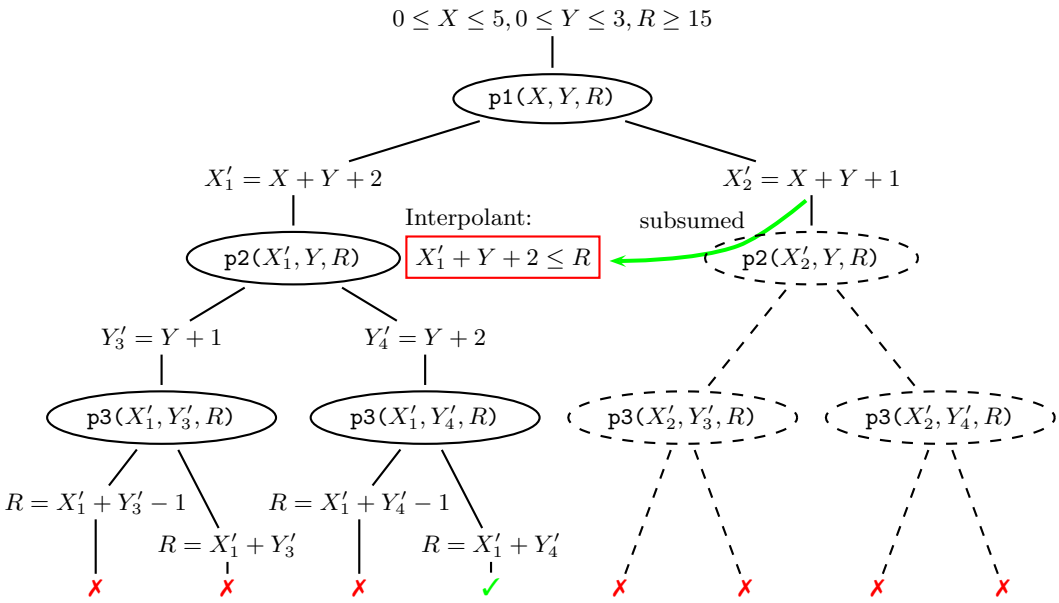

Fig. 2. The FTCLP derivation tree (solid edges) for the program in Figure 1. Dashed parts show the additional parts to make up the CLP tree. The curved arrow shows the shortcut enabled by the interpolant.

`p1`(X_1, Y_1, R_1) produces the renaming $\{X_1 \mapsto X, Y_1 \mapsto Y, R_1 \mapsto R\}$ which transforms the first clause of `p1/3` to: `p1`(X, Y, R) `:-` $X_1' = X + Y + 2$, `p2`(X_1', Y, R) .

The constraint store contains the formula

$$(0 \le X \le 5) \wedge (0 \le Y \le 3) \wedge (R \ge 15) \wedge (X_1' = X+Y+2) \wedge (Y_3' = Y+1) \wedge (R = X_1'+Y_3'-1)$$

which is the conjunction of all primitive constraints in π. Now, this formula is unsatisfiable, so the derivation fails. But before backtracking we want to record some explanation of the failure. This is the role of an *interpolant*.

Given two formulas A and B such that the query $A \wedge B$ is unsatisfiable, an interpolant is a formula I that over-approximates A (that is, $A \models I$) while preserving the falsity of the query (that is, $I \wedge B = \mathit{false}$). The other essential property of an interpolant is that the only variables allowed in I are those common to A and B.

We partition π into A and B where A contains the primitive constraints added up to `p2`(X_1', Y, R), whereas B contains the primitives added subsequently:

$$\begin{aligned} A &= \{0 \le X \le 5, 0 \le Y \le 3, R \ge 15, X_1' = X + Y + 2\} \\ B &= \{Y_3' = Y + 1, R = X_1' + Y_3' - 1\} \end{aligned}$$

By the chosen partitioning, the common variables between A and B are X_1', Y, and R—exactly the variables of the body atom `p2/3` called inside the first clause of `p1/3` (after renaming as explained in Section 2). Interpolants can be computed by most SMT-solvers as a by-product of unsatisfiability proofs. As an example, a valid interpolant computed by MathSAT (Griggio 2012) is $X_1' + Y + 2 \le R$ which can be recorded together with the goal `p2`(X_1', Y, R). The advantage of recording the interpolant I is that whenever we

come across another call to `p2/3` whose current constraints are at least as strong as I, for example at program point /*?*/, we can, without further ado, consider the call failed.

Interpolants are computed for each atom along π (so for $\texttt{p1}(X, Y, R)$ and $\texttt{p3}(X'_1, Y'_3, R)$ as well). We have left them out, as `p2/3` turns out to be the interesting predicate in the example. Moreover, we compute an interpolant for each atom L and for every failed execution within the subtree rooted at L. The interpolant for L is the conjunction of all the interpolants generated from L's subtree. In our example, from the second and third derivations, the interpolants generated for $\texttt{p2}(X'_1, Y, R)$ do not further strengthen the interpolant from the leftmost derivation. Using MathSAT we obtain

$$(X'_1 + Y'_3 + 2 \leq R) \wedge (X'_1 + Y'_3 + 2 \leq R) \wedge (X'_1 + Y'_3 + 2 \leq R) \equiv X'_1 + Y_3 + 2 \leq R$$

Note that the fourth derivation will produce the only answer of the program $X = 5, Y = 3, R = 15$ so no interpolant is generated.

After we backtrack finishing the execution of `p1/3`'s first clause, we would like to avoid exploring the clauses of `p2/3` again when visiting the program point /*?*/. As before, we collect all the constraints up to that point $\pi' \equiv [0 \leq X \leq 5, 0 \leq Y \leq 3, R \geq 15, \texttt{p1}(X, Y, R), X'_2 = X + Y + 1, \texttt{p2}(X'_2, Y, R)]$. To be able to reuse the interpolant from $\texttt{p2}(X'_1, Y, R)$ we must apply the renaming $\{X'_1 \mapsto X'_2\}$. Then, we can check if the current constraint store entails the interpolants stored for $\texttt{p2}(X'_1, Y, R)$ after renaming

$$(0 \leq X \leq 5) \wedge (0 \leq Y \leq 3) \wedge (R \geq 15) \wedge (X'_2 = X + Y + 1) \quad \models \quad X'_2 + Y + 2 \leq R$$

and this entailment is easily seen to hold.

Next we try to reuse the answer computed so far. Like TCLP, FTCLP cannot directly consume answers since an answer may not be possible with the constraint store of the subsumed goal. So when we attempt to reuse an answer, we discard it if it becomes a failure under the execution of the subsumed goal. In our example, the answer from the fourth derivation of Figure 2, effectively, $X'_2 = 10, Y = 3, R = 15$, when renamed and conjoined with the current store, leads to failure. Thus, no more answers are generated.

If the entailment had failed then we would have to re-explore the clauses of `p2/3`. Those explorations generate new interpolants for each clause. Then, we can either form *disjunctive interpolants* or record each one separately as a different entry. The former can provide more pruning than the latter at the expense of more expensive entailment tests. In our experiments, we have implemented both and failed to observe any improvement from keeping disjunctive interpolants. ■

Note that the approach relies, for correctness, on depth-first traversal of the derivation tree. The reuse of an interpolant I for atom H assumes that I has come about as the result of one or more *complete* derivations from H.

It is not hard to see that interpolation corresponds to a weak form of constraint projection, but an interpolant provides unique benefits:

1. it can be computed in linear time relative to the size of the unsatisfiability proof;
2. many useful theories, for which we do not have efficient constraint projection algorithms or any projection at all, are equipped with interpolation: integer and real linear arithmetic, uninterpreted functors, arrays, *etc.*; and
3. it can be quite effective as a reusable condition for pruning other derivations.

$$\langle G' \mid C' \rangle \triangleq \begin{cases} \langle L_2, \ldots, L_m \mid C \wedge L_1 \rangle & \text{if } L_1 \text{ is primitive and } C \wedge L_1 \text{ is satisfiable} \\ \langle L'_1, \ldots, L'_k, L_2, \ldots, L_m \mid C \rangle & \text{if } L_1 \text{ is user-defined and} \\ & (H \text{:-} L'_1, \ldots, L'_k) \in \mathsf{unify}(L_1, \mathit{vars}(C) \cup \mathit{vars}(G)) \\ \langle \Box \mid \mathit{false} \rangle & \text{otherwise} \end{cases}$$

Fig. 3. The result of a derivation step $\langle G \mid C \rangle \Rightarrow \langle G' \mid C' \rangle$ where $G \equiv L_1, \ldots, L_m$.

However, interpolants are computed only from failed derivations, and so our method should not be considered a substitute for projection-based TCLP.

2 Preliminaries

We assume the reader is familiar with the operational semantics of Constraint Logic Programming (CLP) as described by, for example, Marriott and Stuckey (1998). Here we define Craig interpolants, a key concept in our method.

The operational semantics of a CLP program is based on the concept of *derivation*. A *state* σ is a pair written $\langle G \mid C \rangle$ where G is a goal and C is a constraint. A *goal*, G, is a sequence of literals $L_1, \ldots, L_m$ where $m \geq 0$ and each literal is either an atom or a primitive constraint. We assume for simplicity that clauses have been translated so that every atom has *distinct variables* as arguments. In case $m = 0$, we say the goal is *empty*, denoted by the symbol $\Box$. C is called the *constraint store*. A *derivation step* from $\langle G \mid C \rangle$ to $\langle G' \mid C' \rangle$, written $\langle G \mid C \rangle \Rightarrow \langle G' \mid C' \rangle$, is defined in Figure 3. Given two atoms A and A', let $\sigma = \mathsf{variant}(A, A')$ be a renaming such that $\sigma(A) = A'$ or $\sigma = \bot$ if there is no such renaming (the atoms are for a different predicate). The function $\mathsf{unify}(L, W)$ returns the set of all renamed rules originated from matching a selected atom L with the head of a rule renamed to be L and all other variables have been suitably renamed so that all are disjoint from the set W.

A *derivation* for state $\langle G_0 \mid C_0 \rangle$ is the sequence of states $\langle G_0 \mid C_0 \rangle \Rightarrow \langle G_1 \mid C_1 \rangle \Rightarrow \ldots$ such that for each $i \geq 0$ there is a derivation step from $\langle G_i \mid C_i \rangle$ to $\langle G_{i+1} \mid C_{i+1} \rangle$.

A *derivation tree* for a goal G and program P is a tree with states as nodes where each path corresponds to a derivation of G, and branches occur in the tree whenever there is a choice of rule with which to rewrite a user-defined constraint.

Definition 1 (*Craig Interpolant*)
Given formulas A, B with $A \wedge B$ unsatisfiable, a Craig interpolant is a formula P, such that: (1) $A \models P$, (2) $P \wedge B$ is unsatisfiable, and (3) $\mathit{vars}(P) \subseteq \mathit{vars}(A) \cap \mathit{vars}(B)$. ■

An interpolant P allows us to discard irrelevant information from A not needed to ensure unsatisfiability with B. Thus, P is an *over-approximation* of A. Importantly, P is defined only in terms of the variables shared by A and B. In this sense, interpolation acts as a specialized form of projection. Efficient interpolation procedures exist for quantifier-free fragments of theories such as linear real and integer arithmetic, uninterpreted functions, pointers and arrays, and bitvectors. In all these cases, interpolants can be extracted from the refutation proof in time linear in the size of the proof. We refer the reader to Cimatti et al. (2008) and McMillan (2011) for details.

Note that we would like to annotate multiple points along the failed derivation since our ultimate goal is to annotate the whole derivation tree with interpolants. Therefore, it is convenient to use the following definition.

Definition 2 (*Inductive Sequence Interpolant*)
Given a sequence of formulas $\pi \equiv [F_1, \ldots, F_n]$, $[P_0, \ldots, P_n]$ is an *inductive sequence of interpolants* (also called a *path interpolant*) (Jhala and McMillan 2006) for π when:

1. $P_0 = \mathit{true}$ and $P_n = \mathit{false}$,
2. $\forall\ (1 \leq i \leq n) : P_{i-1} \wedge F_i \models P_i$, and
3. $\forall\ (1 \leq i < n) : \mathit{vars}(P_i) \subseteq \mathit{vars}(F_i) \cap \mathit{vars}(F_{i+1})$

That is, the i-th element of the interpolant is a formula in the common language of F_i and F_{i+1}, and is a logical consequence of the first i elements of π. ■

We will assume a procedure SeqIntp that takes a sequence of formulae $[\ F_1, \ldots, F_n\ ]$ and returns an inductive sequence of interpolants $[\ P_0, \ldots, P_n]$. Note that since $P_n = \mathit{false}$ the formula $F_1 \wedge \ldots \wedge F_n$ must be unsatisfiable.

3 A Tabled CLP Algorithm with Interpolation

We present in Figure 4 a new CLP execution algorithm that, given an initial state $\langle G \mid C \rangle$, produces all its answers. During this process, the algorithm explores the derivation tree corresponding to the execution of $\langle G \mid C \rangle$, while recording knowledge about the failed derivations encountered during the traversal as well as all the answers. The main purpose is to eliminate future executions which cannot lead to additional answers. There are two tables at the heart of our algorithm. The *Failure table (FT)* maps an atom A appearing in a derivation to an interpolant (its reuse condition). The *Answer table (AT)* maps an atom A to the set of answers generated so far during the execution of A.

While AT has an entry per predicate, FT has at least one entry per clause head. It is to facilitate this that different clause heads for the same predicate use different variables, as exemplified in Figure 1. The reason for the distinction is that a given predicate can be explored repeatedly during the execution of the program, and we want to store interpolants for each of these explorations separately. Initially, FT maps all entries to $\bot$ and AT maps all entries to $\emptyset$. For clarity of presentation, both tables will be global variables.

Before describing the algorithm, we stress that FailureTabling (as described in Figure 4) will run forever if an infinite derivation is executed. In the next section, we will describe how to extend the algorithm to deal with this difficult problem.

FailureTabling takes two inputs, an initial state $\langle G \mid C \rangle$ and Path, a stack that contains all constraints (both primitive and user-defined) along the current derivation. Its output is the set of answers generated during the execution of $\langle G \mid C \rangle$.

If the goal is empty we simply return a singleton set with the current store as the unique answer. Otherwise if the first literal L_1 is a primitive constraint and satisfiable with the current store, we add it to the store and continue execution. If it is not satisfiable, we update FT with the interpolants generated from the failed derivation.

Generation and combination of interpolants. Since Path is a stack that contains all the constraints along the current derivation, it is a sequence of the form

$$[c_1^1, \ldots, c_{k_1}^1, H_1, c_1^2, \ldots, c_{k_2}^2, H_2, \ldots, c_1^{n-1}, \ldots, c_{k_{n-1}}^{n-1}, H_{n-1}, c_1^n, \ldots, c_{k_n}^n]$$

where c_j^i is a primitive constraint and H_i is an atom. From this, we can form the sequence

$$F = [c_1^1 \wedge \cdots \wedge c_{k_1}^1, c_1^2 \wedge \cdots \wedge c_{k_2}^2, \cdots, c_1^{n-1} \wedge \cdots \wedge c_{k_{n-1}}^{n-1}, c_1^n \wedge \cdots \wedge c_{k_n}^n]$$

```
FailureTabling(⟨G | C⟩, Path)
  if G = □ then return {C}
  Let G be L_1, ..., L_m for some m ≥ 1
  if L_1 is primitive then
    Path.push(L_1)
    if C ∧ L_1 is satisfiable then
      A_ret := FailureTabling(⟨L_2, ..., L_m | C ∧ L_1⟩, Path)
    else
      FT := CombineIntp(FT, Path, SeqIntp(Path))
    Path.pop
  else % L_1 is a user-defined constraint
    if for some H and σ = variant(H, L_1), C ⊨ σ(FT[H]) then
      A_ret := {σ(C') ∧ C | C' ∈ AT(H), σ(C') ∧ C is satisfiable}
      Path.push(not σ(FT[H]))
      FT := CombineIntp(FT, Path, SeqIntp(Path))
      Path.pop
    else
      Rs := unify(L_1, vars(C) ∪ vars(G)))
      Path.push(L_1)
      A_ret := ∅
      foreach L_1 :- L'_1, ..., L'_k in Rs do
        A_ret := A_ret ∪ FailureTabling(⟨L'_1, ..., L'_k, L_2, ..., L_m | C⟩, Path)
      Path.pop
    AT := AT[L_1 ↦ A_ret ∪ AT(L_1)]
  return A_ret
```

Fig. 4. Tabled CLP algorithm based on interpolation.

Then, SeqIntp(F) will return a sequence of inductive interpolants $[P_0, \ldots, P_n]$. From those, $P_1, \ldots, P_{n-1}$ (recall that $P_0 = \mathit{true}$ and $P_n = \mathit{false}$) can be used directly as reuse conditions for atoms $H_1, \ldots, H_{n-1}$, respectively. Abusing notation, the procedure SeqIntp will also take a path constraint Path as input and transform it into a sequence of formulae F, as described above, before calling the interpolation algorithm. Once the interpolants have been generated we need to record them in the failure table. This is the purpose of the procedure CombineIntp that takes as arguments FT, the current path constraints Path, a sequence of interpolants as returned by SeqIntp and returns an updated FT.

```
CombineIntp(FT, Path, [P_1, ..., P_{n-1}])
  foreach H_i in Path do
    if FT(H_i) = ⊥ then FT := FT[H_i ↦ true]
    FT := FT[H_i ↦ P_i ∧ FT(H_i)]
  return FT
```

Pruning and reusing answers. When we reach a user-defined constraint L_1 we first check whether the failure table has an answer for atom H that can be reused. If the current store C implies the suitably renamed interpolant $FT[H]$, we filter the answers for H to find those applicable to the current state $\langle G \mid C\rangle$. In general we may have multiple entries in FT that are variants of L_1. Thus, we check each of these entries to see if the entailment test succeeds. Moreover, we need to generate interpolants from the subsumed derivation. Otherwise, it would be unsound. Note that we know that $C \models \sigma(FT[H])$ or equivalently, $C \wedge \mathit{not}\ \sigma(FT[H])$ is unsatisfiable. Note also that Path contains all constraints in C but it does not contain $\mathit{not}\ \sigma(FT[H])$. So we simply push $\mathit{not}\ \sigma(FT[H])$ temporarily onto Path and call SeqIntp. As before, the new interpolants must be combined by the procedure CombineIntp which will further update FT. Otherwise we consider all the

rules that match L_1 and recursively visit the resulting states, collecting all the answers in A_{ret}. When we pop L_1 off the path, we are guaranteed that the interpolant stored in $FT[L_1]$ is correct, since it then considers all possible derivations for L_1.[1] Finally, before leaving the call to FailureTabling we update the answers attached to L_1 in the answer table AT to include A_{ret}.

We can show that FailureTabling returns the correct answers for a state, by showing that when the FT lookup succeeds, it returns the correct answers for the derivation.

Theorem 1
Given a goal G, the algorithm in Figure 4 preserves all answers. That is,

$$\text{FailureTabling}(\langle G \mid true\rangle, [\,]) = \{C \mid \langle G \mid true\rangle \Rightarrow^* \langle \square \mid C\rangle,\ C \text{ is satisfiable}\} \quad \square$$

We observed during the evaluation of our method (discussed in Section 5) that if we generate an interpolant for each atom within a failed derivation, the algorithm described in Figure 4 can degrade. The reason is that the number of calls to the interpolation procedure is too high. Thus, it is important to reduce the number of calls to interpolation. The following lemma provides a key optimization.

Lemma 1
Let $\pi_1 = [F_1, \ldots, F_k, F_{k+1}, \ldots, F_n]$ and $\pi_2 = [F_1, \ldots, F_k, F'_{k+1}, \ldots, F'_m]$ be two formula sequences, and let $[P_0, \ldots, P_n]$ and $[P'_0, \ldots P'_m]$ be two corresponding inductive sequence interpolants. If $P_k \models P'_k$ then $[P_0, \ldots, P_k, P'_{k+1}, \ldots, P'_m]$ is a correct inductive sequence interpolant for π_2. $\square$

This allows us to start computing interpolants backwards from P'_m in a lazy manner and stop if for some point i, $P_i \models P'_i$ where P_i is the current interpolant at that point, without the need to eagerly compute all the interpolants from each failed derivation.

4 How to Handle Infinite Derivations

The algorithm FailureTabling does not give any special treatment to recursive clauses. Therefore, it will not terminate in the presence of infinite derivations. We propose a simple extension based on *counter instrumentation* (Gulwani et al. 2009; McMillan 2010) that can produce complete and finite derivation trees even with infinite derivations:

1. Transform the original program P into a new program P' by adding a new counter variable k_i to each recursive clause i such that k_i is decremented by one each time a goal is called recursively. Moreover, each recursive clause fails if $k_i < 0$. The purpose of this transformation is to generate some *fake* failed derivations (depending on k_i counters) within recursive clauses so that the CLP execution can terminate.
2. Set each $d_i = 0$ (these are the limits on recursive calls for clause i)
3. Run FailureTabling on P' with $k_i = d_i$
4. After termination of FailureTabling extract from each failed derivation the deepest atom L that corresponds to a recursive clause i and whose interpolant involves k_i. Then, check if the procedure Check_Inductive_Invariant(L,i) succeeds for every L. If yes, we can stop. Otherwise, set $d_i = d_i + 1$ and go to Step 3.

[1] Here we assume that there are no two syntactically identical recursive calls—renaming can ensure this.

$x := i$; $y := j$;	`?-` $X = I, Y = J,$ `1(`X, Y, I, J`)` .
while $(x \neq 0)$ {	
$x := x - 1$;	`1(`X_1, Y_1, I_1, J_1`)` `:-` $X_1 \neq 0, X_1' = X_1 - 1, Y_1' = Y_1 - 1,$
$y := y - 1$;	`1(`X_1', Y_1', I_1, J_1`)` .
}	`1(`X_2, Y_2, I_2, J_2`)` `:-` $X_2 = 0,$ `error(`X_2, Y_2, I_2, J_2`)` .
if ($i == j$) **assert**$(y \leq 0)$;	`error(`$\_, Y_3, I_3, J_3$`)` `:-` $I_3 = J_3, Y_3 > 0$.

Fig. 5. C and CLP version of a verification example from (Jhala and McMillan 2006).

We now describe how the procedure CHECK_INDUCTIVE_INVARIANT(L,i) performs the inductive invariant test. During the execution of FAILURETABLING we keep track, for every failed derivation, of the atom L at level d_i that corresponds to a recursive clause and whose interpolant may depend on k_i (that is, the interpolant is fake). Moreover, we identify each ancestor L_{anc} of L. For a given derivation π in which L appears, the set of all its ancestors is all the occurrences of the same recursive predicate corresponding to L that is defined above on π (that is, at level $< d_i$).

For every atom L we then repeat the following process until there are no more candidates to check (we have at most d_i candidates for each L) or a candidate is confirmed to be an inductive invariant for every L. Let $I_{L_{anc}}$ be the interpolant (our candidate) of an ancestor L_{anc} after ignoring the part that involves k_i. This can be done by plugging $k_i = 0$. Then check that $I_{L_{anc}}$ in conjunction with all the constraints from every possible derivation from L_{anc} to L *entails* the candidate $I_{L_{anc}}$ after proper renaming.

This process is the analogue of tabling's completion check, but without the need for constraint projection. Note that our renaming to perform the entailment test is purely syntactic and it does not need projection.

We omit details of the entailment test because although straightforward it is quite tedious. Instead, we show our method for handling infinite derivations through an example.

Example 2 (*Infinite derivations*)
CLP has been shown a successful model for verifying *safety* properties in *infinite state systems*, see for example Jaffar et al. (2009), Angelis et al. (2012). The resulting Horn clauses are usually recursive and the goal is to prove that CLP model of the program is empty (that is, no answers). Figure 5 shows a C program from Jhala and McMillan (2006) and its corresponding CLP translation. We describe now how our method can terminate and prove that the derivation tree has no successful derivations.

We first show the code of the recursive predicate after the counter instrumentation-based transformation that ensures a finite derivation tree:

$$\begin{aligned}
&\texttt{1}(X_1, Y_1, I_1, J_1, K_1) \texttt{ :- } K_1 \geq 0,\ X_1 \neq 0,\ X_1' = X_1 - 1, Y_1' = Y_1 - 1, K_1' = K_1 - 1,\\
&\qquad\qquad \texttt{1}(X_1', Y_1', I_1, J_1, K_1') \ .\\
&\texttt{1}(X_2, Y_2, I_2, J_2, _) \texttt{ :- } X_2 = 0,\ \texttt{error}(X_2, Y_2, I_2, J_2) \ .
\end{aligned}$$

Let us assume we fix $d_i = 0$ and let us focus on the interpolants generated for the atom `1(`X, Y, I, J, K`)`. From one derivation (the one corresponding to the recursive clause) we obtain $K \leq 0$ and from the derivation that explores `error/4` (the non-recursive clause) we obtain the interpolant $Y + I \leq X + J$. Therefore, the resulting interpolant is $P \equiv K \leq 0 \wedge Y + I \leq X + J$. Unfortunately, we cannot claim to prove the derivation tree has no solutions yet since the interpolant depends on K which was not originally in the

		CLP			FTCLP			FTCLP+opt
C	Answers	Time(s)	States	Failure	Time(s)	States	Failure	Time(s)
150	2	1.7	5112	1056	4.3	4269	862	3.8
200	16	14.4	40342	8302	22.5	16870	3026	18.8
225	58	44.1	116684	23961	47.3	28624	4636	37.5
250	164	138	323154	65865	88.2	43796	6327	67.0
275	451	450	827770	167969	148.6	61276	7529	111.5

Table 1. *Execution of a* CLP *program that implements the RCSP problem with data from the instance rcsp-1 in the OR-library. All answers with cost* $\leq$ C *are generated.*

program. We then construct the formula $P' \equiv (P \wedge K = 0)$ to eliminate the dependency on K. Next we check whether P' is an *inductive invariant*. That is,

$$\overbrace{(Y_1 + I_1 \leq X_1 + J_1)}^{C} \wedge \overbrace{(X_1 \neq 0, X_1' = X_1 - 1, Y_1' = Y_1 - 1)}^{\Pi} \models \overbrace{(Y_1' + I_1 \leq X_1' + J_1)}^{C'}$$

Note that we rename both C and C' using substitutions $\{X \mapsto X_1, Y \mapsto Y_1, I \mapsto I_1, J \mapsto J_1\}$ and $\{X \mapsto X_1', Y \mapsto Y_1', I \mapsto I_1, J \mapsto J_1\}$, respectively but once again constraint projection is not required. In general, Π can be a disjunctive formula encoding the execution of all body literals. Although this may be expensive, we rely on SMT to deal with it. Since the entailment test holds, we have proven that the interpolant $P' \equiv Y + I \leq X + J$ is an inductive invariant and hence, we can finally claim that the derivation tree will not have answers, that is, the safety property holds. Note that the invariant cannot be expressed in difference logic. Therefore, a TCLP system such as that of Chico de Guzmán et al. (2012) using a difference logic constraint domain (which has efficient projection) will run forever. It is worth mentioning that, in general, the interpolant generated may not be an inductive invariant. In that case, we increment the value of d_i and repeat the process (for example, the program t5 in the online appendix requires to fix $d_i = 2$). The process of repeatedly incrementing d_i could, of course, also be an indefinite one. ■

5 Experimental Evaluation

To evaluate our method we have implemented a proof-of-concept CLP meta-interpreter[2] using the Ciao system (Hermenegildo et al. 2012) and the SMT solver MathSAT (Griggio 2012) for checking satisfiability and generation of interpolants. Our prototype does not necessarily compute inductive sequence interpolants. This does not affect the use of Theorem 1 but it does affect the applicability of Lemma 1. Thus, we only make use of Lemma 1 after checking that a sequence interpolant is inductive.

All experiments[3] have been run on a single core of a 2.7GHz Core i7-26202M with 8GB memory. The first experiment compares CLP programs that implement the RCSP problem examined in Jaffar et al. (2008). The results are shown in Table 1. Columns labelled with CLP, FTCLP, and FTCLP+opt is our interpreter without pruning, with pruning, and with pruning and optimization using Lemma 1, respectively. Clearly as the size of derivation

[2] Available at `http://code.google.com/p/ftclp` together with all test programs.
[3] It would be natural to compare against the system of Chico de Guzmán et al. (2012). However, at the time of writing, this system is not entirely stable, and we observed incorrect behaviour in the two experiments.

Program	FTCLP	BLAST	HSF	TRACER
t1	∞	∞	0.3s	∞
t1-a	0.1s	ERR	0.2s	∞
t2	0.1s	0.1s	UNSAFE	0.1s
t3	0.1s	0.1s	0.25s	0.1s
t4	0.1s	0.6s	0.3s	0.2s
t5	0.2s	0.2s	0.3s	0.1s

Table 2. *Comparing* FTCLP *with several verifiers for some verification problems. For* FTCLP *we show execution time of running the* CLP *encodings. For the rest, we show the time to verify the C programs. ERR indicates an error,* ∞ *timeout after 5 minutes, and UNSAFE is a false positive.*

tree (column States) grows, the number of failed derivations (column Failure) grows and failure tabling becomes more and more competitive. The overhead of computing and looking up interpolants eventually pays off in terms of a massive search reduction.

Although the results of this experiment are promising, we recognize that the execution times obtained by FTCLP+opt are not yet fully satisfactory. In a preliminary experiment we modified our interpreter to use the simplex method implemented in the Ciao system and compared with the same interpreter with MathSAT. We observed that the former was at least one order of magnitude faster than the latter. We suspect that the method implemented in Ciao is more incremental that the one in MathSAT.

The second experiment compares the verification times of several C programs, each taken from different verification publications. In the interest of the readers, we give each original C program and its CLP encoding in the online appendix. Table 2 shows the execution times for FTCLP in running the CLP encodings. There are no successful derivations for any of these programs (since the conditions to be verified holds in each case). We also show the verification times of three state-of-the-art verifiers: BLAST (Beyer et al. 2007), HSF (Grebenshchikov et al. 2012) and TRACER (Jaffar et al. 2012) taking the C programs as inputs. Both HSF and TRACER use CLP as intermediate representation. As can be seen, FTCLP is highly robust, and comparable to specialised verification tools on these problems, while it can achieve termination in cases where other tools cannot.

6 Related Work

In previous sections we compared with TCLP through several examples We review now works from other areas (mostly from verification) which inspired us.

Jaffar et al. (2008) proposed a solution for a combinatorial optimization problem, the Resource-Constrained Shortest Path (RCSP). The RCSP problem is modelled as a CLP program and a dynamic programming algorithm computes either *strongest* or *weakest preconditions* for pruning derivations that violate the maximum resource consumption. Since it is an optimization problem the method keeps track of only one answer, namely the optimal one. Jaffar et al. (2009) focus on the problem of verification of safety properties in loop-free C programs.[4] The C program is again modelled as a CLP program, and then verification boils down to checking whether the program does not have any successful

[4] Although the method can be extended to handle loops assuming loop invariants are provided.

derivation. The method only prunes *finite* derivations and since this is a decision problem, no answers are generated. Our method can be seen as a generalization of these works. Rather than solving specific problems we aim at improving arbitrary (recursive) CLP programs. Another important difference is that Jaffar et al. (2008) and Jaffar et al. (2009) generate reuse conditions by computing either strongest or weakest preconditions which are defined in terms of constraint projection.

Another approach for proving safety properties computes an over-approximation of the set of reachable states via *predicate abstraction* (Graf and Saïdi 1997). The set of predicates π is usually very coarse (for example, $\pi = \emptyset$) and new predicates are added into π whenever a counterexample is found. Interpolation has proven a very effective technique to discover those predicates (Henzinger et al. 2004). Gupta et al. (2011a) observed that the interpolation problem can be reduced to solving a set of recursion-free *Horn clauses*. They proposed an algorithm that builds a derivation tree proving the unsatisfiability of a set of recursion-free Horn clauses and augments it with some inference rules to compute interpolants over linear rational arithmetic. Gupta et al. (2011b) and Grebenshchikov et al. (2012) follow this line of research and extend it for uninterpreted functions and further applications such as inter-procedural interpolants. A very recent work also using CLP clauses as intermediate representation is Rümmer et al. (2013) which introduces *disjunctive interpolants* solving a more general class of problems in one step by handling multiple paths simultaneously.

These works encode into CLP clauses only the set of spurious counterexamples encountered by the predicate abstraction-based exploration. Since these counterexamples are spurious (*i.e.* unsatisfiable) and finite, the set of CLP clauses is ensured to be recursion-free and without answers. Moreover, they rely on building a derivation tree which can be of exponential size relative to the number of clauses since no pruning techniques are employed. Our approach can be also seen as an interpolation method based on Horn clauses and, in fact, it subsumes previous works since it can handle (recursive) Horn clauses producing also disjunctive interpolants without restriction (*e.g.* Rümmer et al. 2013 handle only *body disjoint* Horn clauses). Moreover, these methods can benefit from our pruning technique to mitigate their exponential search space.

7 Conclusions and Future Work

We have presented a new CLP execution strategy called Failure Tabling (FTCLP) that allows the pruning of redundant failed derivations, and can produce, in some cases, a finite derivation tree even in the presence of infinite executions. From the verification community we have borrowed ideas developed for symbolic reachability with interpolation and we have adapted these to the new setting of executing CLP programs. Interpolation can remove the tyrannic dependency in TCLP on projection algorithms which may either not exist or be too inefficient. However, FTCLP should not be seen as a substitute for TCLP since they provide different benefits.

Future work should involve better assessing of the practical benefits of FTCLP with a broader set of programs, the generation of inductive sequence interpolants (*e.g.* Christ et al. 2012), and the integration of our method within a TCLP system (such as Chico de Guzmán et al. 2012) to take advantage of a real tabling implementation.

References

Angelis, E. D., Fioravanti, F., Proietti, M., and Pettorossi, A. 2012. Software model checking by program specialization. In *CILC*. 89–103.

Beyer, D., Henzinger, T., Jhala, R., and Majumdar, R. 2007. The software model checker BLAST. *International Journal on Software Tools for Technology Transfer (STTT) 9*, 505–525.

Chico de Guzmán, P., Carro, M., Hermenegildo, M. V., and Stuckey, P. J. 2012. A general implementation framework for tabled CLP. In *FLOPS*. 104–119.

Christ, J., Hoenicke, J., and Nutz, A. 2012. SMTInterpol: An interpolating SMT solver. In *SPIN*. 248–254.

Cimatti, A., Griggio, A., and Sebastiani, R. 2008. Efficient interpolant generation in satisfiability modulo theories. In *TACAS*. 397–412.

Codognet, P. 1995. A tabulation method for constraint logic programming. In *Symposium and Exhibition on Industrial Applications of Prolog*.

Craig, W. 1957. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic 22,* 3, 250–268.

Graf, S. and Saïdi, H. 1997. Construction of abstract state graphs with pvs. In *CAV*. 72–83.

Grebenshchikov, S., Lopes, N. P., Popeea, C., and Rybalchenko, A. 2012. Synthesizing software verifiers from proof rules. In *PLDI*. 405–416.

Griggio, A. 2012. A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. *Journal on Satisfiability, Boolean Modeling and Computation 8*, 1–27.

Gulwani, S., Mehra, K. K., and Chilimbi, T. M. 2009. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*. 127–139.

Gupta, A., Popeea, C., and Rybalchenko, A. 2011a. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*. 331–344.

Gupta, A., Popeea, C., and Rybalchenko, A. 2011b. Solving recursion-free Horn clauses over LI+UIF. In *APLAS*. 188–203.

Henzinger, T. A., Jhala, R., Majumdar, R., and McMillan, K. L. 2004. Abstractions from proofs. In *POPL*. 232–244.

Hermenegildo, M. V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J. F., and Puebla, G. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming 12,* 1–2, 219–252.

Jaffar, J. and Lassez, J. 1987. Constraint logic programming. In *POPL*. 111–119.

Jaffar, J., Murali, V., Navas, J. A., and Santosa, A. E. 2012. TRACER: A symbolic execution tool for verification. In *CAV*. 758–766.

Jaffar, J., Santosa, A. E., and Voicu, R. 2008. Efficient memoization for dynamic programming with ad-hoc constraints. In *AAAI*. 297–303.

Jaffar, J., Santosa, A. E., and Voicu, R. 2009. An interpolation method for CLP traversal. In *CP*. 454–469.

Jhala, R. and McMillan, K. L. 2006. A practical and complete approach to predicate refinement. In *TACAS*. 459–473.

Marriott, K. and Stuckey, P. J. 1998. *Introduction to Constraint Logic Programming*. MIT Press, Cambridge, MA, USA.

McMillan, K. L. 2010. Lazy annotation for program testing and verification. In *CAV*. 104–118.

McMillan, K. L. 2011. Interpolants from z3 proofs. In *FMCAD*. 19–27.

Rümmer, P., Hojjat, H., and Kuncak, V. 2013. Disjunctive interpolants for horn-clause verification. In *CAV*. 347–363.

Tamaki, H. and Sato, T. 1986. OLD resolution with tabulation. In *ICLP*. 84–98.

Warren, D. S. 1992. Memoing for logic programs. *Communcations of the ACM 35,* 3, 93–111.