# Disjunctive Interpolants for Horn-Clause Verification

Philipp Rümmer[1], Hossein Hojjat[2], and Viktor Kuncak[2]

[1] Uppsala University, Sweden
[2] Swiss Federal Institute of Technology Lausanne (EPFL)

**Abstract.** One of the main challenges in software verification is efficient and precise compositional analysis of programs with procedures and loops. Interpolation methods remains one of the most promising techniques for such verification, and are closely related to solving Horn clause constraints. We introduce a new notion of interpolation, disjunctive interpolation, which solves a more general class of problems in one step compared to previous notions of interpolants, such as tree interpolants or inductive sequences of interpolants. We present algorithms and complexity for construction of disjunctive interpolants, as well as their use within an abstraction-refinement loop. We have implemented Horn clause verification algorithms that use disjunctive interpolants and evaluate them on benchmarks expressed as Horn clauses over the theory of integer linear arithmetic.

## 1 Introduction

Software model checking has greatly benefited from the combination of a number of seminal ideas: automated abstraction through theorem proving [8], exploration of finite-state abstractions, and counterexample-driven refinement [3]. Even though these techniques can be viewed independently, the effectiveness of verification has been consistently improving by providing more sophisticated communication between these steps. Often, carefully chosen search aspects are being pushed into a learning-enabled constraint solver, resulting in better overall verification performance. An essential advance was to use interpolants derived from unsatisfiability proofs to refine the abstraction [13]. In recent years, we have seen significant progress in interpolating methods for different logical constraints [4, 5, 21], and a wealth of more general forms of interpolation [1, 12, 21, 24]. In this paper we identify a new notion, *disjunctive interpolants*, which are more general than tree interpolants and inductive sequences of interpolants. Like tree interpolation [12, 21], a disjunctive interpolation query is a tree-shaped constraint specifying the interpolants to be derived; however, in disjunctive interpolation, branching in the tree can represent both conjunctions and disjunctions. We present an algorithm for solving the interpolation problem, relating it to a subclass of recursion-free Horn clauses [10, 22, 23]. We then consider solving general recursion-free Horn clauses and show that this problem is solvable whenever the logic admits interpolation. We establish tight complexity bounds for solving recursion-free Horn clauses for propositional logic (PSPACE) and for integer linear arithmetic (co-NEXPTIME). In contrast, the disjunctive interpolation problem remains in coNP for these logics. We also show how to use solvers for recursion-free Horn clauses to verify recursive Horn clauses using counterexample-driven predicate abstraction. We present an algorithm and experimental results on publicly available benchmarks.

### 1.1 Related Work

There is a long line of research on Craig **interpolation** methods, and generalised forms of interpolation tailored to verification. For an overview of interpolation in the presence of theories, we refer the reader to [4, 5]. Binary Craig interpolation for implications $A \to C$ goes back to [6], was used on conjunctions $A \wedge B$ in [19], and generalised to inductive sequences of interpolants in [13, 20]. The concept of tree interpolation, strictly generalising inductive sequences of interpolants, is presented in the documentation of the interpolation engine iZ3 and in [21]; the computation of tree interpolants by computing a sequence of binary interpolants is also described in [12]. In this paper, we present a new form of interpolation, *disjunctive interpolation*, which is strictly more general than sequences of interpolants and tree interpolants. Our implementation supports Presburger arithmetic, including divisibility constraints [4], which is rarely supported by existing tools, yet helpful in practice [15].

A further generalisation of inductive sequences of interpolants are restricted DAG interpolants [1], which also include disjunctiveness in the sense that multiple paths through a program can be handled simultaneously. Disjunctive interpolants are incomparable in power to restricted DAG interpolants, since the former does not handle interpolation problems in the form of DAGs, while the latter does not subsume tree interpolation. A combination of the two kinds of interpolants ("disjunctive DAG interpolation") is strictly more powerful (and harder) than disjunctive interpolation, see Sect. 5.1 for a complexity-theoretic analysis. We discuss techniques and heuristics to practically handle shared sub-trees in disjunctive interpolation, extending the benefits of DAG interpolation to recursive programs.

Inter-procedural **software model checking** with interpolants has been an active area of research. In the context of predicate abstraction, it has been discussed how well-scoped invariants can be inferred [13] in the presence of function calls. Based on the concept of Horn clauses, a predicate abstraction-based algorithm for bottom-up construction of function summaries was presented in [9]. Encoding into Horn clauses is also used in logic programming community [23]. Verification of programs with procedures is described in [12] (using nested word automata) as well as in [2]. Function summaries generated using interpolants have also been used in bounded model checking [26]. Researchers also showed how to lift these techniques to higher-order programs [17, 28].

The use of **Horn clauses** as intermediate representation for verification was proposed in [10], with the verification of concurrent programs as main application. The underlying procedure for solving sets of recursion-free Horn clauses, over the combined theory of linear *rational* arithmetic and uninterpreted functions, was presented in [11]. An algorithm to solve recursion-free systems of Horn constraints by repeated computation of binary interpolants was given in [27], for the purpose of type inference. A range of further applications of Horn clauses, including inter-procedural model checking, was given in [9]. Horn clauses are also used as a format for verification problems supported by the SMT solver Z3 [14]. Our paper extends this direction by presenting general results about solvability and computational complexity, independent of any particular calculus. Our experiments are with linear *integer* arithmetic, arguably a more faithful model of discrete computation than rationals [15].

(1) $\mathsf{gcd}(M,N,R) \leftarrow M = N \wedge R = M$

(2) $\mathsf{gcd}(M,N,R) \leftarrow M > N \wedge M1 = M - N \wedge \mathsf{gcd}(M1,N,R)$

(3) $\mathsf{gcd}(M,N,R) \leftarrow M < N \wedge N1 = N - M \wedge \mathsf{gcd}(M,N1,R)$

(4) $\mathsf{false} \leftarrow M \geq 0 \wedge M = N \wedge \mathsf{gcd}(M,N,R) \wedge R > M$

**Fig. 1.** Horn clauses computing the greatest common divisor of two numbers and an assertion on result. Variables are universally quantified in each clause.

(1) $\mathsf{gcd}(M,N,R) \leftarrow M = N \wedge R = M$

(1') $\mathsf{gcd1}(M,N,R) \leftarrow M = N \wedge R = M$

(2') $\mathsf{gcd}(M,N,R) \leftarrow M > N \wedge M1 = M - N \wedge \mathsf{gcd1}(M1,N,R)$

(3') $\mathsf{gcd}(M,N,R) \leftarrow M < N \wedge N1 = N - M \wedge \mathsf{gcd1}(M,N1,R)$

(4) $\mathsf{false} \leftarrow M \geq 0 \wedge M = N \wedge \mathsf{gcd}(M,N,R) \wedge R > M$

**Fig. 2.** Extended recursion-free approximation of the Horn clauses in Fig. 1.

## 2 Example: Verification of Recursive Predicates

We start by showing how our approach can verify programs encoded as Horn clauses, by means of predicate abstraction and a theorem prover for Presburger arithmetic. Fig. 1 shows an example of a system of Horn clauses that compute the greatest common divisor of its first and its second argument in its third argument. After invoking the gcd operation on the equal positive numbers $M$ and $N$, we wish to check whether it is possible for the result $R$ to be more than the $M$. In general, we encode error conditions as Horn clauses with *false* in their head, and refer to such clauses as error clauses, although such clauses do not have a special semantic status in our system. When executed with these clauses as input, our verification tool automatically identifies that the definition of gcd(M,N,R) as the predicate $(M = N) \rightarrow (M \geq R)$ gives a solution to these Horn clauses. In terms of safety (partial correctness), this means that the error condition cannot be reached.

Our approach uses counterexample-driven refinement to perform verification. In this example, the abstraction of Horn clauses starts with a trivial set of predicates, containing only the predicate *false*, which is assumed to be a valid approximation until proven otherwise. Upon examining a clause that has a concrete satisfiable formula on the right-hand side (e.g. $M = N \wedge R = M$), we rule out *false* as the approximation of gcd. In the absence of other candidate predicates, the approximation of gcd becomes the conjunction of an empty set of predicates, which is *true*. Using this approximation the error clause is no longer satisfied. At this point the algorithm checks whether a true error is reached by directly chaining the clauses involved in computing the approximation of predicates. This amounts to checking whether the following recursion-free subset of clauses has a solution:

(1) $\mathsf{gcd}(M,N,R) \leftarrow M = N \wedge R = M$

(4) $\mathsf{false} \leftarrow M \geq 0 \wedge M = N \wedge \mathsf{gcd}(M,N,R) \wedge R > M$

The solution to above problem is any formula $I(M,N,R)$ such that

$\mathsf{I(M,N,R)} \leftarrow \mathsf{M = N} \wedge \mathsf{R = M}$
$\mathsf{false} \quad \leftarrow \mathsf{M} \geq 0 \wedge \mathsf{M = N} \wedge \mathsf{I(M,N,R)} \wedge \mathsf{R > M}$

This is precisely an interpolant of $M = N \wedge R = M$ and $M \geq 0 \wedge M = N \wedge R > M$. A valid interpolant is $P_1(M,N,R) \equiv M \geq R$. Choosing this interpolant eliminates the current contradiction for Horn clauses and $P_1$ is added into a list of abstraction predicates for the relation gcd. Because the predicates approximating gcd are now updated, we consider the abstraction of the system in terms of these predicates.

The predicate $P_1$ is not a conjunct in a valid approximation for gcd in clause (2), so the following recursion-free unfolding is not solved by the approximation so far:

(1) $\mathsf{gcd(M,N,R)} \leftarrow \mathsf{M = N} \wedge \mathsf{R = M}$
(2') $\mathsf{gcd1(M,N,R)} \leftarrow \mathsf{M > N} \wedge \mathsf{M1 = M - N} \wedge \mathsf{gcd(M1,N,R)}$
(4') $\mathsf{false} \quad \leftarrow \mathsf{M} \geq 0 \wedge \mathsf{M = N} \wedge \mathsf{gcd1(M,N,R)} \wedge \mathsf{R > M}$

This particular problem could be reduced to solving an interpolation sequence, but it is more natural to think of it simply as a solution for recursion-free Horn clauses. A solution is an interpretation of the relations gcd and gcd1 as ternary relations on integers, such that the clauses are true. Note that this problem could also be viewed as the computation of tree interpolants, which are also a special case of solving recursion-free Horn clauses, as are DAG interpolants and a new notion of disjunctive tree interpolants that we introduce. In line with [9–11] we observe that recursion-free clauses are a perfect fit for counterexample-driven verification: they allow us to provide the theorem proving procedure with much more information that they can use to refine abstractions. In the limit, the original set of clauses or its recursive unfoldings are its own approximations, some of them exact, but the advantage of *recursion-free* Horn clauses is that their solvability is decidable under very general conditions. This provides us with a solid theorem proving building block to construct robust and predictable solvers for the undecidable recursive case. Our paper describes a new such building block: disjunctive interpolants, which correspond to a subclass of non-recursive Horn clauses.

To illustrate disjunctive interpolants, Fig. 2 provides another recursion-free approximations of the problem. In this approximation we can distinguish 3 different paths from the error clause (4) through the clauses (1'), (2') and (3') to ground formulae. The traditional refinement approach using e.g. tree interpolation typically removes the 3 instances of the spurious counter-examples using 3 interpolation calls. A novelty of disjunctive interpolation is removing the different choices of counter-examples altogether using a single call to the interpolating theorem prover. Eliminating more counter-examples at once can reduce the number of iterations and increase convergence.

## 3 Formulae and Horn Clauses

*Constraint languages.* Throughout this paper, we assume that a first-order vocabulary of *interpreted symbols* has been fixed, consisting of a set $\mathcal{F}$ of fixed-arity function symbols, and a set $\mathcal{P}$ of fixed-arity predicate symbols. Interpretation of $\mathcal{F}$ and $\mathcal{P}$ is determined by a class $\mathcal{S}$ of structures $(U, I)$ consisting of non-empty universe $U$, and a mapping $I$ that assigns to each function in $\mathcal{F}$ a set-theoretic function over $U$, and to each predicate in $\mathcal{P}$ a set-theoretic relation over $U$. As a convention, we assume the presence of an equation symbol "=" in $\mathcal{P}$, with the usual interpretation. Given a countably infinite set $\mathcal{X}$ of variables, a *constraint language* is a set *Constr* of first-order formulae over $\mathcal{F}, \mathcal{P}, \mathcal{X}$ For example, the language of quantifier-free Presburger arithmetic has $\mathcal{F} = \{+, -, 0, 1, 2, \ldots\}$ and $\mathcal{P} = \{=, \leq, |\}$).

A constraint is called *satisfiable* if it holds for some structure in $\mathcal{S}$ and some assignment of the variables $\mathcal{X}$, otherwise *unsatisfiable*. We say that a set $\Gamma \subseteq \mathit{Constr}$ of constraints *entails* a constraint $\phi \in \mathit{Constr}$ if every structure and variable assignment that satisfies all constraints in $\Gamma$ also satisfies $\phi$; this is denoted by $\Gamma \models \phi$.

$\mathit{fv}(\phi)$ denotes the set of free variables in constraint $\phi$. We write $\phi[x_1, \ldots, x_n]$ to state that a constraint contains (only) the free variables $x_1, \ldots, x_n$, and $\phi[t_1, \ldots, t_n]$ for the result of substituting the terms $t_1, \ldots, t_n$ for $x_1, \ldots, x_n$. Given a constraint $\phi$ containing the free variables $x_1, \ldots, x_n$, we write $\mathit{Cl}_\forall(\phi)$ for the *universal closure* $\forall x_1, \ldots, x_n.\phi$.

*Positions.* We denote the set of *positions* in a constraint $\phi$ by *positions*($\phi$). For instance, the constraint $a \wedge \neg a$ has 4 positions, corresponding to the sub-formulae $a \wedge \neg a$, $\neg a$, and the two occurrences of $a$. The sub-formula of a formula $\phi$ underneath a position $p$ is denoted by $\phi \downarrow p$, and we write $\phi[p/\psi]$ for the result of replacing the sub-formula $\phi \downarrow p$ with $\psi$. Further, we write $p \leq q$ if position $p$ is above $q$ (that is, $q$ denotes a position within the sub-formula $\phi \downarrow p$), and $p < q$ if $p$ is strictly above $q$.

*Craig interpolation* is the main technique used to construct and refine abstractions in software model checking. A binary interpolation problem is a conjunction $A \wedge B$ of constraints. A *Craig interpolant* is a constraint $I$ such that $A \models I$ and $B \models \neg I$, and such that $\mathit{fv}(I) \subseteq \mathit{fv}(A) \cap \mathit{fv}(B)$. The existence of an interpolant implies that $A \wedge B$ is unsatisfiable. We say that a constraint language has the *interpolation property* if also the opposite holds: whenever $A \wedge B$ is unsatisfiable, there is an interpolant $I$.

### 3.1 Horn Clauses

To define the concept of Horn clauses, we fix a set $\mathcal{R}$ of uninterpreted fixed-arity *relation symbols,* disjoint from $\mathcal{P}$ and $\mathcal{F}$. A *Horn clause* is a formula $C \wedge B_1 \wedge \cdots \wedge B_n \to H$ where

- $C$ is a constraint over $\mathcal{F}, \mathcal{P}, \mathcal{X}$;
- each $B_i$ is an application $p(t_1, \ldots, t_k)$ of a relation symbol $p \in \mathcal{R}$ to first-order terms over $\mathcal{F}, \mathcal{X}$;
- $H$ is similarly either an application $p(t_1, \ldots, t_k)$ of $p \in \mathcal{R}$ to first-order terms, or is the constraint *false*.

$H$ is called the *head* of the clause, $C \wedge B_1 \wedge \cdots \wedge B_n$ the *body*. In case $C = \mathit{true}$, we usually leave out $C$ and just write $B_1 \wedge \cdots \wedge B_n \to H$. First-order variables (from $\mathcal{X}$) in a clause are considered implicitly universally quantified; relation symbols represent set-theoretic relations over the universe $U$ of a structure $(U, I) \in \mathcal{S}$. Notions like (un)satisfiability and entailment generalise straightforwardly to formulae with relation symbols.

A *relation symbol assignment* is a mapping $\mathit{sol} : \mathcal{R} \to \mathit{Constr}$ that maps each $n$-ary relation symbol $p \in \mathcal{R}$ to a constraint $\mathit{sol}(p) = C_p[x_1, \ldots, x_n]$ with $n$ free variables. The *instantiation* $sol(h)$ of a Horn clause $h$ is defined by:

$$\begin{aligned} sol(C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow p(\bar{t})) &= C \wedge sol(p_1)[\bar{t}_1] \wedge \cdots \wedge sol(p_n)[\bar{t}_n] \rightarrow sol(p)[\bar{t}] \\ sol(C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow \mathit{false}) &= C \wedge sol(p_1)[\bar{t}_1] \wedge \cdots \wedge sol(p_n)[\bar{t}_n] \rightarrow \mathit{false} \end{aligned}$$

**Definition 1 (Solvability).** *Let $\mathcal{HC}$ be a set of Horn clauses over relation symbols $\mathcal{R}$.*

1. *$\mathcal{HC}$ is called semantically solvable if for every structure $(U, I) \in \mathcal{S}$ there is an interpretation of the relation symbols $\mathcal{R}$ as set-theoretic relations over $U$ such that the universally quantified closure $Cl_\forall(h)$ of every clause $h \in \mathcal{HC}$ holds in $(U, I)$.*
2. *$\mathcal{HC}$ is called syntactically solvable if there is a relation symbol assignment sol such that for every structure $(U, I) \in \mathcal{S}$ and every clause $h \in \mathcal{HC}$ it is the case that $Cl_\forall(sol(h))$ is satisfied.*

Note that, in the special case when $\mathcal{S}$ contains only one structure, $\mathcal{S} = \{(U, I)\}$, semantic solvability reduces to the existence of relations interpreting $\mathcal{R}$ that extend the structure $(U, I)$ in such a way to make all clauses true. In other words, Horn clauses are solvable in a structure if and only if the extension of the theory of $(U, I)$ by relation symbols $\mathcal{R}$ in the vocabulary and by given Horn clauses as axioms is consistent.

Clearly, if a set of Horn clauses is syntactically solvable, then it is also semantically solvable. The converse is not true in general, because the solution need not be expressible in the constraint language (see Appendix E of [25] for an example).

A set $\mathcal{HC}$ of Horn clauses induces a *dependence relation* $\rightarrow_{\mathcal{HC}}$ on $\mathcal{R}$, defining $p \rightarrow_{\mathcal{HC}} q$ if there is a Horn clause in $\mathcal{HC}$ that contains $p$ in its head, and $q$ in the body. The set $\mathcal{HC}$ is called *recursion-free* if $\rightarrow_{\mathcal{HC}}$ is acyclic, and *recursive* otherwise. In the next sections we study the solvability problem for recursion-free Horn clauses; in particular, Theorem 2 below characterises the relationship between syntactic and semantic solvability for recursion-free Horn clauses. This case is relevant, since solvers for recursion-free Horn clauses form a main component of many general Horn-clause-based verification systems [9, 10].

## 4 Disjunctive Interpolants and Body-Disjoint Horn Clauses

Having defined the classical notions of interpolation and Horn clauses, we now present our notion of disjunctive interpolants, and the corresponding class of Horn clauses. Our inspiration are generalized forms of Craig interpolation, such as inductive sequences of interpolants [13, 20] or tree interpolants [12, 21]. We introduce disjunctive interpolation as a new form of interpolation that is tailored to the refinement of abstractions in Horn clause verification, strictly generalising both inductive sequences of interpolants and tree interpolation. Disjunctive interpolation problems can specify both conjunctive and disjunctive relationships between interpolants, and are thus applicable for simultaneous analysis of multiple paths in a program, but also tailored to inter-procedural analysis or verification of concurrent programs [9].

Disjunctive interpolation problems correspond to a specific fragment of recursion-free Horn clauses, namely recursion-free body-disjoint Horn clauses (see Sect. 4.1). The definition of disjunctive interpolation is chosen deliberately to be as general as possible, while still avoiding the high computational complexity of solving general systems of recursion-free Horn clauses. Computational complexity is discussed in Sect. 5.1.

We introduce disjunctive interpolants as a form of *sub-formula abstraction*. For example, given an unsatisfiable constraint $\phi[\alpha]$ containing $\alpha$ as a sub-formula in a positive position, the goal is to find an abstraction $\alpha'$ such that $\alpha \models \alpha'$ and $\alpha[\alpha'] \models \mathit{false}$, and such that $\alpha'$ only contains variables common to $\alpha$ and $\phi[\mathit{true}]$. Generalizing this to any number of subformulas, we obtain the following.

**Definition 2 (Disjunctive interpolant).** *Let $\phi$ be a constraint, and $\mathit{pos} \subseteq \mathit{positions}(\phi)$ a set of positions in $\phi$ that are only underneath the connectives $\wedge$ and $\vee$. A disjunctive interpolant is a map $I : \mathit{pos} \to \mathit{Constr}$ from positions to constraints such that:*

1. *For each position $p \in \mathit{pos}$, with direct children*
   $\{q_1, \ldots, q_n\} \;=\; \{q \in \mathit{pos} \mid p < q \text{ and } \neg\exists r \in \mathit{pos}.\ p < r < q\}$ *we have*
   $$(\phi[q_1/I(q_1), \ldots, q_n/I(q_n)]){\downarrow}p \;\models\; I(p)\ ,$$
2. *For the topmost positions $\{q_1, \ldots, q_n\} \;=\; \{q \in \mathit{pos} \mid \neg\exists r \in \mathit{pos}.\ r < q\}$ we have*
   $$\phi[q_1/I(q_1), \ldots, q_n/I(q_n)] \;\models\; \mathit{false}\ ,$$
3. *For each position $p \in \mathit{pos}$, we have $\mathit{fv}(I(p)) \subseteq \mathit{fv}(\phi{\downarrow}p) \cap \mathit{fv}(\phi[p/\mathit{true}])$.*

*Example 1.* Consider $A_p \wedge B$, with position $p$ pointing to the sub-formula $A$, and $\mathit{pos} = \{p\}$. The disjunctive interpolants for $A \wedge B$ and $\mathit{pos}$ coincide with the ordinary binary interpolants for $A \wedge B$.

*Example 2.* Consider the formula $\phi = (\cdots(((T_1)_{p_1} \wedge T_2)_{p_2} \wedge T_3)_{p_3} \wedge \cdots)_{p_{n-1}} \wedge T_n$ and positions $\mathit{pos} = \{p_1, \ldots, p_{n-1}\}$. Disjunctive interpolants for $\phi$ and $\mathit{pos}$ correspond to inductive sequences of interpolants [13,20]. Note that we have the entailments
$T_1 \models I(p_1),\ I(p_1) \wedge T_2 \models I(p_2),\ \ldots,\ I(p_{n-1}) \wedge T_n \models \mathit{false}.$

*Example 3.* Tree interpolation problems correspond to disjunctive interpolation with a set $\mathit{pos}$ of positions that are only underneath $\wedge$ (and never underneath $\vee$). We give a precise definition and results about the existence of tree interpolants in [24].

*Example 4.* We consider the example given in Fig. 2, Sect. 2. To compute a solution for the Horn clauses, we first *expand* the Horn clauses into a constraint, by means of exhaustive inlining/resolution (see Sect. 5), obtaining a disjunctive interpolation problem:

$$\begin{aligned}
\mathit{false} \;&\rightsquigarrow\; M \geq 0 \wedge M = N \wedge \mathsf{gcd}(M, N, R) \wedge R > M \\
&\rightsquigarrow\; \left(\begin{array}{r} M \geq 0 \\ \wedge\, M = N \\ \wedge\, R > M \end{array}\right) \wedge \left(\begin{array}{c} M = N \wedge R = M \\ \vee \\ M > N \wedge M_1 = M - N \wedge \mathsf{gcd1}(M_1, N, R) \\ \vee \\ M < N \wedge N_1 = N - M \wedge \mathsf{gcd1}(M, N_1, R) \end{array}\right) \\
&\rightsquigarrow\; \left(\begin{array}{r} M \geq 0 \\ \wedge\, M = N \\ \wedge\, R > M \end{array}\right) \wedge \left(\begin{array}{c} M = N \wedge R = M \\ \vee \\ M > N \wedge M_1 = M - N \wedge (M_1 = N \wedge R = M_1)_q \\ \vee \\ M < N \wedge N_1 = N - M \wedge (M = N_1 \wedge R = M)_r \end{array}\right)_p
\end{aligned}$$

In the last formula, the positions $p, q, r$ corresponding to the relation symbol $\mathsf{gcd}$ and the two occurrences of $\mathsf{gcd1}$ are marked. It can be observed that the last formula is unsatisfiable, and that $I = \{p \mapsto ((M = N) \to (M \geq R)),\ q \mapsto \mathit{true},\ r \mapsto \mathit{true}\}$ is a disjunctive interpolant. A solution for the Horn clauses can be derived from the interpolant by conjoining the constraints derived for the two occurrences of $\mathsf{gcd1}$:

$$\mathsf{gcd}(M, N, R) \;=\; ((M = N) \to (M \geq R)), \quad \mathsf{gcd1}(M, N, R) \;=\; \mathit{true}$$

**Theorem 1.** *Suppose $\phi$ is a constraint, and suppose $pos \subseteq positions(\phi)$ is a set of positions in $\phi$ that are only underneath the connectives $\wedge$ and $\vee$. If Constr is a constraint language that has the interpolation property, then a disjunctive interpolant I exists for $\phi$ and pos if and only if $\phi$ is unsatisfiable.*

*Proof.* "⇒" By means of simple induction, we can derive that $\phi \downarrow p \models I(p)$ holds for every disjunctive interpolant $I$ for $\phi$ and $pos$, and for every $p \in pos$. From Def. 2, it then follows that $\phi$ is unsatisfiable.

"⇐" Suppose $\phi$ is unsatisfiable. We encode the disjunctive interpolation problem into a (conjunctive) tree interpolation problem (following the terminology in [24]) by adding auxiliary Boolean variables.[3] Wlog, we assume that $pos$ contains the root position $root$ of $\phi$. The graph of the tree interpolation problem is $(pos, E)$, with the edge relation $E = \{(p, q) \mid p < q \text{ and } \neg\exists r. p < r < q\}$. For every $p \in pos$, let $a_p$ be a fresh Boolean variable. We label the nodes of the tree using the function $\phi_L : pos \to Constr$. For each position $p \in pos$, with direct children $\{q_1, \ldots, q_n\} = \{q \in pos \mid E(p, q)\}$ we define

$$\phi_L(p) \;=\; \begin{cases} \phi[q_1/a_{q_1}, \ldots, q_n/a_{q_n}] & \text{if } p = root \\ \neg a_p \vee (\phi[q_1/a_{q_1}, \ldots, q_n/a_{q_n}]) \downarrow p & \text{otherwise} \end{cases}$$

Observe that $\bigwedge_{p \in pos} \phi_L(p)$ is unsatisfiable. According to [24], a tree interpolant $I_T$ exists for this labelling function. By construction, for non-root positions $p \in pos \setminus \{root\}$ the interpolant labelling is equivalent to $I_T(p) \equiv \neg a_p \vee I_p$, where $I_p$ does not contain any further auxiliary Boolean variables. We can then construct a disjunctive interpolant $I$ for the original problem as

$$I(p) \;=\; \begin{cases} \mathit{false} & \text{if } p = root \\ I_p & \text{otherwise} \end{cases}$$

To see that $I$ is a disjunctive interpolant, observe that for each position $p \in pos$ with direct children $\{q_1, \ldots, q_n\} = \{q \in pos \mid E(p, q)\}$ the following entailment holds (since $I_T$ is a tree interpolant): $\phi_L(p) \wedge (\neg a_{q_1} \vee I_{q_1}) \wedge \cdots \wedge (\neg a_{q_n} \vee I_{q_n}) \models I_T(p)$
Via Boolean reasoning this implies: $(\phi[q_1/I_{q_1}, \ldots, q_n/I_{q_n}]) \downarrow p \models I(p)$. □

The proof provides a constructive method to solve disjunctive interpolation problems, by means of transformation to a tree interpolation problem. This is also the algorithm that we used in our experiments in Sect. 6.2; practical aspects of this approach are discussed in the beginning of Sect. 6.

[3] The concept of auxiliary Boolean variables to represent interpolation problems has also been used in [26] and [2], for the purpose of extracting function summaries in model checking.

### 4.1 Solvability of Body-Disjoint Horn Clauses

The relationship between Craig interpolation and (syntactic) solutions of Horn clauses has been observed in [11]. Disjunctive interpolation corresponds to a specific class of recursion-free Horn clauses, namely Horn clauses that are *body disjoint:*

**Definition 3.** *A finite, recursion-free set $\mathcal{HC}$ of Horn clauses is body disjoint if for each relation symbol $p$ there is at most one clause containing $p$ in its body, and every clause contains $p$ at most once.*

An example for body-disjoint clauses is the subset $\{(1), (4)\}$ of clauses in Fig. 1. Syntactic solutions of a set $\mathcal{HC}$ of body-disjoint Horn clauses can be computed by solving a disjunctive interpolation problem; vice versa, every disjunctive interpolation problem can be translated into an equivalent set of body-disjoint clauses.

In order to extract an interpolation problem from $\mathcal{HC}$, we first normalise the clauses: for every relation symbol $p \in \mathcal{R}$, we fix a unique vector of variables $\bar{x}_p$, and rewrite $\mathcal{HC}$ such that $p$ only occurs in the form $p(\bar{x}_p)$. This is possible due to the fact that $\mathcal{HC}$ is body disjoint. The translation from Horn clauses to a disjunctive interpolation problem is done recursively, similar in spirit to inlining of function invocations in a program; thanks to body-disjointness, the encoding is polynomial.

$$
\begin{aligned}
\mathit{enc}(\mathcal{HC}) &= \bigvee_{(C \wedge B_1 \wedge \cdots \wedge B_n \to \mathit{false}) \in \mathcal{HC}} C \wedge \mathit{enc}'(B_1) \wedge \cdots \wedge \mathit{enc}'(B_n) \\
\mathit{enc}'(p(\bar{x}_p)) &= \left( \bigvee_{(C \wedge B_1 \wedge \cdots \wedge B_n \to p(\bar{x}_p)) \in \mathcal{HC}} C \wedge \mathit{enc}'(B_1) \wedge \cdots \wedge \mathit{enc}'(B_n) \right)_{l_p}
\end{aligned}
$$

Note that the resulting formula $\mathit{enc}(\mathcal{HC})$ contains a unique position $l_p$ at which the definition of a relation symbol $p$ is inlined; in the second equation, this position is marked with $l_p$. Any disjunctive interpolant $I$ for this set of positions represents a syntactic solution of $\mathcal{HC}$, and vice versa.

## 5 Solvability of Recursion-free Horn Clauses

The previous section discussed how the class of recursion-free body-disjoint Horn clauses can be solved by reduction to disjunctive interpolation. We next show that this construction can be generalised to arbitrary systems of recursion-free Horn clauses. In absence of the body-disjointness condition, however, the encoding of Horn clauses as interpolation problems can incur a potentially exponential blowup. We give a complexity-theoretic argument justifying that this blowup cannot be avoided in general. This puts disjunctive interpolation (and, equivalently, body-disjoint Horn clauses) at a sweet spot: preserving the relatively low complexity of ordinary binary Craig interpolation, while carrying much of the flexibility of the Horn clause framework.

We first introduce the exhaustive *expansion* $\mathit{exp}(\mathcal{HC})$ of a set $\mathcal{HC}$ of Horn clauses, which generalises the Horn clause encoding from the previous section. We write $C' \wedge B'_1 \wedge \cdots \wedge B'_n \to H'$ for a fresh variant of a Horn clause $C \wedge B_1 \wedge \cdots \wedge B_n \to H$, i.e., the clause obtained by replacing all free first-order variables with fresh variables. Expansion is then defined by the following recursive functions:

$$\begin{aligned} \mathit{exp}(\mathcal{HC}) &= \bigvee_{(C \wedge B_1 \wedge \cdots \wedge B_n \to \mathit{false}) \in \mathcal{HC}} C' \wedge \mathit{exp}'(B_1') \wedge \cdots \wedge \mathit{exp}'(B_n') \\ \mathit{exp}'(p(\bar{t})) &= \bigvee_{(C \wedge B_1 \wedge \cdots \wedge B_n \to p(\bar{s})) \in \mathcal{HC}} C' \wedge \mathit{exp}'(B_1') \wedge \cdots \wedge \mathit{exp}'(B_n') \wedge \bar{t} = \bar{s}' \end{aligned}$$

Note that *exp* is only well-defined for finite and recursion-free sets of Horn clauses, since the expansion might not terminate otherwise.

**Theorem 2 (Solvability of recursion-free Horn clauses).** *Let $\mathcal{HC}$ be a finite, recursion-free set of Horn clauses. If the underlying constraint language has the interpolation property, then the following statements are equivalent:*

1. *$\mathcal{HC}$ is semantically solvable;*
2. *$\mathcal{HC}$ is syntactically solvable;*
3. *$\mathit{exp}(\mathcal{HC})$ is unsatisfiable.*

*Proof.* $2 \Rightarrow 1$ holds because a syntactic solution gives rise to a semantic solution by interpreting the solution constraints. $\neg 3 \Rightarrow \neg 1$ holds because a model of $\mathit{exp}(\mathcal{HC})$ witnesses domain elements that every semantic solution of $\mathcal{HC}$ has to contain, but which violate at least one clause of the form $C \wedge B_1 \wedge \cdots \wedge B_n \to \mathit{false}$, implying that no semantic solution can exist. $3 \Rightarrow 2$ is shown by encoding $\mathcal{HC}$ into a disjunctive interpolation problem (Sect. 4), which can solved with the help of Theorem 1. To this end, clauses are first duplicated to obtain a problem that is body disjoint, and subsequently normalised as described in Sect. 4.1. More details are given in Appendix A of [25]. □

### 5.1 The Complexity of Recursion-free Horn Clauses

Theorem 2 gives rise to a general algorithm for (syntactically) solving recursion-free sets $\mathcal{HC}$ of Horn clauses, over constraint languages for which interpolation procedures are available. The general algorithm requires, however, to generate and solve the expansion $\mathit{exp}(\mathcal{HC})$ of the Horn clauses, which can be exponentially bigger than $\mathcal{HC}$ (in case $\mathcal{HC}$ is not body disjoint), and might therefore require exponential time. This leads to the question whether more efficient algorithms are possible for solving Horn clauses.

We give a number of complexity results about (semantic) Horn clause solvability; proofs of the results are given in the Appendix of [25]. Most importantly, we can observe that solvability is PSPACE-hard, for every non-trivial constraint language *Constr*. The authors of [18] conjecture a similar complexity result for the case of programs with procedures.

**Lemma 1.** *Suppose a constraint language can distinguish at least two values, i.e., there are two ground terms $t_0$ and $t_1$ such that $t_0 \neq t_1$ is satisfiable. Then the semantic solvability problem for recursion-free Horn clauses is PSPACE-hard.*

Looking for upper bounds, it is easy to see that solvability of Horn clauses is in co-NEXPTIME for any constraint language with satisfiability problem in NP (for instance, quantifier-free Presburger arithmetic). This is because the size of the expansion $exp(\mathcal{HC})$ is at most exponential in the size of $\mathcal{HC}$. Individual constraint languages admit more efficient solvability checks:

**Theorem 3.** *Semantic solvability of recursion-free Horn clauses over the constraint language of Booleans is PSPACE-complete.*

Constraint languages that are more expressive than the Booleans lead to a significant increase in the complexity of solving Horn clauses. The lower bound in the following theorem can be shown by simulating time-bounded non-deterministic Turing machines.

**Theorem 4.** *Semantic solvability of recursion-free Horn clauses over the constraint language of quantifier-free Presburger arithmetic is co-NEXPTIME-complete.*

The lower bounds in Lemma 1 and Theorem 4 hinge on the fact that sets of Horn clauses can contain shared relation symbols in bodies. Neither result holds if we restrict attention to body-disjoint Horn clauses, which correspond to disjunctive interpolation as introduced in Sect. 4. Since the expansion $exp(\mathcal{HC})$ of body-disjoint Horn clauses is linear in the size of the set of Horn clauses, also solvability can be checked efficiently:

**Theorem 5.** *Semantic solvability of a set of body-disjoint Horn clauses, and equivalently the existence of a solution for a disjunctive interpolation problem, is in co-NP when working over the constraint languages of Booleans and quantifier-free Presburger arithmetic.*

Body-disjoint Horn clauses are still expressive: they can directly encode acyclic control-flow graphs, as well as acyclic unfolding of many simple recursion patterns.

For proofs of all results of this section, please consult [25].

## 6 Model Checking with Recursive Horn Clauses

Whereas *recursion-free* Horn clauses generalise the concept of Craig interpolation, solving *recursive* Horn clauses corresponds to the verification of general programs with loops, recursion, or concurrency features [9]. Procedures to solve recursion-free Horn clauses can serve as a building block within model checking algorithms for recursive Horn clauses [9], and are used to construct or refine abstractions by analysing spurious counterexamples. In particular, our disjunctive interpolation can be used for this purpose, and offers a high degree of flexibility due to the possibility to analyse counterexamples combining multiple execution traces. We illustrate the use of disjunctive interpolation within a predicate abstraction-based algorithm for solving Horn clauses. Our model checking algorithm is similar in spirit to the procedure in [9], and is explained in Sect. 6.1.

*And/or trees of clauses.* For sake of presentation, in our algorithm we represent counterexamples (i.e., recursion-free sets of Horn clauses) in the form of and/or trees labelled with clauses. Such trees are defined by the following grammar:

$$\mathit{AOTree} \ ::= \ \mathit{And}(h, \mathit{AOTree}, \ldots, \mathit{AOTree}) \ \mid \ \mathit{Or}(\mathit{AOTree}, \ldots, \mathit{AOTree})$$

where $h$ ranges over (possibly recursive) Horn clauses. We only consider well-formed trees, in which the children of every *And*-node have head symbols that are consistent with the body literals of the clause stored in the node, and the sub-trees of an *Or*-node all have the same head symbol. And/or trees are turned into body-disjoint recursion-free sets of clauses by renaming relation symbols appropriately.

*Example 5.* The clauses in Fig. 2 can be represented by the following and/or tree (referring to clauses in Fig. 1).

$$\mathit{And}\Big((4),\ \mathit{Or}(\mathit{And}((1)),\ \mathit{And}((2),\ \mathit{And}((1))),\ \mathit{And}((3), \mathit{And}((1))))\Big)$$

*Solving and/or dags.* Counterexamples extracted from model checking problems often assume the form of and/or *dags*, rather than and/or *trees*. Since and/or-dags correspond to Horn clauses that are not body-disjoint, the complexity-theoretic results of the last section imply that it is in general impossible to avoid the expansion of and/or-dags to and/or-trees; there are, however, various effective techniques to speed-up handling of and/or-dags (related to the techniques in [18]). We highlight two of the techniques we use in our interpolation engine Princess [4], which we used in our experimental evaluation of the next section:

*1) counterexample-guided expansion* expands and/or-dags lazily, until an unsatisfiable fragment of the fully expanded tree has been found; such a fragment is sufficient to compute a solution. Counterexamples are useful in two ways: they can determine which or-branch of an and/or-dag is still satisfiable and has to be expanded further, but also whether it is necessary to create further copies of a shared subtree.

*2) and/or dag restructuring* factors out common sub-dags underneath an *Or*-node, making the and/or-dag more tree-like.

### 6.1 A Predicate Abstraction-based Model Checking Algorithm

Our model checking algorithm is in Fig. 3, and similar in spirit as the procedure in [9]; it has been implemented in the model checker Eldarica.[4] Solutions for Horn clauses are constructed in disjunctive normal form by building an abstract reachability graph over a set of given predicates. When a counterexample is detected (a clause with consistent body literals and head *false*), a theorem prover is used to verify that the counterexample is genuine; spurious counterexamples are eliminated by generating additional predicates by means of disjunctive interpolation.

In Fig. 3, $\Pi : \mathcal{R} \to \mathcal{P}_{\mathrm{fin}}(\mathit{Constr})$ denotes a mapping from relation symbols to the current (finite) set of predicates used to approximate the relation symbol. Given a (possibly recursive) set $\mathcal{HC}$ of Horn clauses, we define an *abstract reachability graph* (ARG) as a hyper-graph $(S, E)$, where

[4] http://lara.epfl.ch/w/eldarica

- $S \subseteq \{(p, Q) \mid p \in \mathcal{R}, Q \subseteq \Pi(p)\}$ is the set of nodes, each of which is a pair consisting of a relation symbol and a set of predicates.
- $E \subseteq S^* \times \mathcal{HC} \times S$ is the hyper-edge relation, with each edge being labelled with a clause. An edge $E(\langle s_1, \ldots, s_n \rangle, h, s)$, with $h = (C \wedge B_1 \wedge \cdots \wedge B_n \to H) \in \mathcal{HC}$, implies that
  - $s_i = (p_i, Q_i)$ and $B_i = p_i(\bar{t}_i)$ for all $i = 1, \ldots, n$, and
  - $s = (p, Q)$, $H = p(\bar{t})$, and $Q = \{\phi \in \Pi(p) \mid C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \models \phi[\bar{t}]\}$, where we write $Q_i[\bar{t}_i]$ for the conjunction of the predicates $Q_i$ instantiated for the argument terms $t_i$.

An ARG $(S, E)$ is called *closed* if the edge relation represents all Horn clauses in $\mathcal{HC}$. This means, for every clause $h = (C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \to H) \in \mathcal{HC}$ and every sequence $(p_1, Q_1), \ldots, (p_n, Q_n) \in S$ of nodes one of the following properties holds:

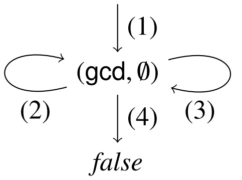
- $C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \models \mathit{false}$, or
- there is an edge $E(\langle (p_1, Q_1), \ldots, (p_n, Q_n) \rangle, C, s)$ such that $s = (p, Q)$, $H = p(\bar{t})$, and $Q = \{\phi \in \Pi(p) \mid C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \models \phi[\bar{t}]\}$.

**Lemma 2.** *A set $\mathcal{HC}$ of Horn clauses has a closed ARG $(S, E)$ if and only if $\mathcal{HC}$ is syntactically solvable.*

A proof is given in Appendix F of [25]. The function ExtractCEX extracts an and/or-tree representing a set of counterexamples, which can be turned into a recursion-free body-disjoint set of Horn clauses, and solved as described in Sect. 4.1. In general, the tree contains both conjunctions (from clauses with multiple body literals) and disjunctions, generated when following multiple hyper-edges (the case $|T| > 1$). Disjunctions make it possible to eliminate multiple counterexamples simultaneously. The algorithm is parametric in the precise strategy used to compute counterexamples (represented as non-deterministic choice in the pseudo code). The strategies we evaluated in the experiments (shown in the next section) are:

**TI** extraction of a single counterexamples with minimal depth
(which means that disjunctive interpolation reduces to **T**ree **I**nterpolation), and
**DI** simultaneous extraction of all counterexamples with minimal depth
(so that genuine **D**isjunctive **I**nterpolation is used).

*Example 6.* We consider the Horn clauses given in Fig. 1, Sect. 2. Starting with an empty predicate map $\Pi$, the function ConstructARG will construct the reachability graph shown on the right (edges are labelled with the clauses from Fig. 1). Since *false* is reachable, function ExtractCEX will be called to extract a counterexample; possible results of executing ExtractCEX include:



$$tree_1 = And((4),\ And((1))),$$
$$tree_2 = And((4),\ Or(And((1)),\ And((2),And((1))),\ And((3),And((1)))))$$

The counterexample $tree_2$ corresponds to the clauses shown in Fig. 2. Elimination of this counterexample with the help of disjunctive interpolation yields the predicates discussed in Example 4, which are sufficient to construct a closed ARG.

```
$S := \emptyset,\ E := \emptyset,\ \Pi := \{p \mapsto \emptyset \mid p \in \mathcal{R}\}$        ▷ Empty graph, no predicates
function ConstructARG
    while $true$ do
        pick clause $h = (C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \to H) \in \mathcal{HC}$
            and nodes $(p_1, Q_1), \ldots, (p_n, Q_n) \in S$
            such that $\neg\exists s.\ (\langle (p_1, Q_1), \ldots, (p_n, Q_n)\rangle, h, s) \in E$
            and $C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \not\models false$
        if no such clauses and nodes exist then return $\mathcal{HC}$ is solvable
        if $H = false$ then                                                    ▷ Refinement needed
            $tree := And(h, \text{ExtractCEX}(p_1, Q_1), \ldots, \text{ExtractCEX}(p_n, Q_n)$
            if $tree$ is unsatisfiable then
                extract disjunctive interpolant from $tree$, add predicates to $\Pi$
                delete part of $(S, E)$ used to construct $tree$
            else return $\mathcal{HC}$ is unsolvable, with counterexample trace $tree$
        else                                                                   ▷ Add edge to ARG
            then $H = p(\bar{t})$
            $Q := \{\phi \in \Pi(p) \mid \{C\} \cup Q_1 \cup \ldots \cup Q_n \models \phi\}$
            $e := (\langle (p_1, Q_1), \ldots, (p_n, Q_n)\rangle, h, (p, Q))$
            $S := S \cup \{(p, Q)\},\ E := E \cup \{e\}$

function ExtractCEX($root : S$)                         ▷ Extract disjunctive interpolation problem
    pick $\emptyset \neq T \subseteq E$ with $\forall e \in T.\ e = (\_, \_, root)$
    return $Or\{\, And(h, \text{ExtractCEX}(s_1), \ldots, \text{ExtractCEX}(s_n)) \mid (\langle s_1, \ldots, s_n\rangle, h, root) \in T\, \}$
```

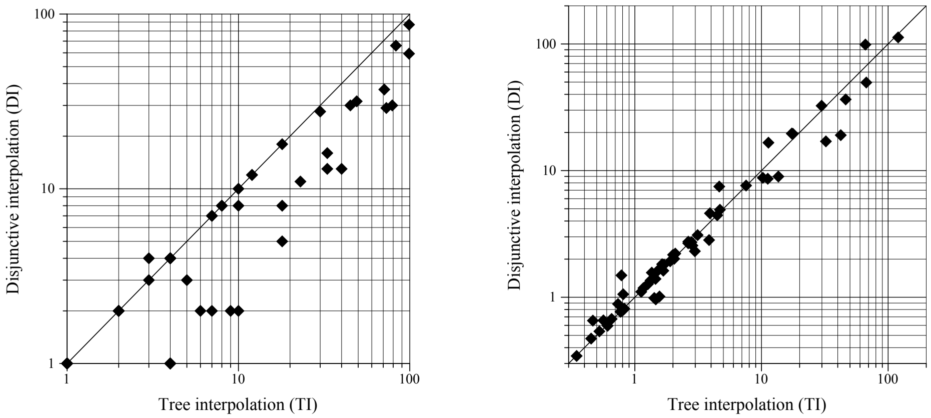
**Fig. 3.** Algorithm for construction of abstract reachability graphs.

We remark that we have also implemented a simpler "global" algorithm that approximates each relation symbol globally with a single conjunction of inferred predicates instead of disjunction of conjunctions. The two algorithms behave similarly in our experience, with the global one occasionally slower, but conceptually simpler. What allowed us to use a simpler algorithm is precisely the more general form of the interpolation. This shows another advantage of more expressive interpolation: the simplicity of verification algorithms we can build on top of it.

### 6.2 Experimental Evaluation

We have evaluated our algorithm on a set of benchmarks in integer linear arithmetic from the NTS library [16] translated into Horn clauses [5]. These include recursive algorithms, benchmarks extracted from programs with singly-linked lists, VHDL models of circuits, verification conditions for programs with arrays, benchmarks from the NECLA static analysis suite, and C programs with asynchronous procedure calls translated using the approach of [7]. Scatter plots comparing the results for the **T**ree **I**nterpolation and **D**isjunctive **I**nterpolation runs are given in Fig. 4. A table with detailed data is provided in [25]. The experiments show comparable verification times and performance for tree interpolation and disjunctive interpolation runs. Studying the results more closely, we

[5] https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/LIA/



(a) Number of CEGAR iterations

(b) Model checking runtime

**Fig. 4.** Comparison of the number of required refinement steps, and the runtime (in seconds), for the case of single counterexamples (**TI**) and simultaneous extraction of all minimal-depth counterexamples (**DI**). All experiments were done on an Intel Core i5 2-core machine with 3.2GHz and 8Gb, with a timeout of 900s.

observed that **DI** consistently led to a smaller number of abstraction refinement steps (the scatter plot in Fig. 4); this indicates that **DI** is indeed able to eliminate multiple counterexamples simultaneously, and to rapidly generate predicates that are useful for abstraction. The experiments also showed that there is a trade-off between the time spent generating predicates, and the quality of the predicates. In **TI**, on average 31% of the verification is used for predicate generation (interpolation), while with **DI** 42% is used; in some of the benchmarks from [7], this led to the phenomenon that **DI** was slower than **TI**, despite fewer refinement steps. This may change as we make further improvements to our prototype implementation of disjunctive interpolation. We also compared our results to the performance of HSF,[6] a state-of-the-art verification engine for Horn clauses. HSF was faster on average, but for harder examples [7] our tool was comparable (see the technical report for detailed results).

## Conclusions

We have introduced disjunctive interpolation as a new form of Craig interpolation tailored to model checkers based on Horn clauses. Disjunctive interpolation can be identified as solving body-disjoint systems of recursion-free Horn clauses, and subsumes a number of previous forms of interpolation, including tree interpolation. We believe that the flexibility of disjunctive interpolation is highly beneficial for building interpolation-based model checkers. We expect further performance improvements from better implementation of disjunctive interpolation and better techniques to select sets of counterexample paths given to interpolation.

[6] http://www7.in.tum.de/tools/hsf/

## References

1. A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig interpretation. In *SAS*, 2012.
2. A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In *VMCAI*, pages 39–55, 2012.
3. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS'02*, volume 2280 of *LNCS*, page 158, 2002.
4. A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. *Journal of Automated Reasoning*, 47:341–367, 2011.
5. A. Cimatti, A. Griggio, and R. Sebastiani. Efficient generation of Craig interpolants in satisfiability modulo theories. *ACM Trans. Comput. Log.*, 12(1):7, 2010.
6. W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *The Journal of Symbolic Logic*, 22(3):250–268, September 1957.
7. P. Ganty and R. Majumdar. Algorithmic verification of asynchronous programs. *CoRR*, abs/1011.0551, 2010.
8. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV*, 1997.
9. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.
10. A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, 2011.
11. A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free horn clauses over LI+UIF. In *APLAS*, pages 188–203, 2011.
12. M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, 2010.
13. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244. ACM, 2004.
14. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.
15. H. Hojjat, R. Iosif, F. Konečný, V. Kuncak, and P. Rümmer. Accelerating interpolants. In *Automated Technology for Verification and Analysis (ATVA)*, 2012.
16. H. Hojjat, F. Konečný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A verification toolkit for numerical transition systems (tool paper). In *FM*, 2012.
17. R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: Verifying functional programs using abstract interpreters. In *CAV*, 2011.
18. A. Lal, S. Qadeer, and S. K. Lahiri. Corral: A solver for reachability modulo theories. In *CAV*, 2012.
19. K. L. McMillan. Interpolation and SAT-based model checking. In *CAV*, 2003.
20. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.
21. K. L. McMillan and A. Rybalchenko. Solving constrained Horn clauses using interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, Jan. 2013.
22. M. Méndez-Lojo, J. A. Navas, and M. V. Hermenegildo. A flexible, (C)LP-based approach to the analysis of object-oriented programs. In *LOPSTR*, pages 154–168, 2007.
23. J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. In *SAS*, 1998.
24. P. Rümmer, H. Hojjat, and V. Kuncak. Classifying and solving horn clauses for verification. In *VSTTE*, 2013.
25. P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for horn-clause verification (extended technical report). *CoRR*, abs/1301.4973, 2013.
26. O. Sery, G. Fedyukovich, and N. Sharygina. Interpolation-based function summaries in bounded model checking. In *Haifa Verification Conference*, pages 160–175, 2011.
27. T. Terauchi. Dependent types from counterexamples. In M. V. Hermenegildo and J. Palsberg, editors, *POPL*, pages 119–130. ACM, 2010.
28. H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In *POPL*, 2013.