Guiding Craig interpolation with domain-specific abstractions

Jérôme Leroux, Philipp Rümmer, Pavle Subotić

Noname manuscript No.
(will be inserted by the editor)

Guiding Craig Interpolation with Domain-specific Abstractions

Jérôme Leroux · Philipp Rümmer · Pavle Subotić

Received: date / Accepted: date

Abstract Craig Interpolation is a standard method to construct and refine abstractions in model checking. To obtain abstractions that are suitable for the verification of software programs or hardware designs, model checkers rely on theorem provers to find the right interpolants, or interpolants containing the right predicates, in a generally infinite lattice of interpolants for any given interpolation problem. We present a semantic and solver-independent framework for systematically exploring interpolant lattices, based on the notion of *interpolation abstraction.* We discuss how interpolation abstractions can be constructed for a variety of logics, and how they can be applied in the context of software model checking.

Keywords Craig Interpolation · Abstraction · Model Checking

1 Introduction

Model checkers use abstractions to reduce the state space of software programs or hardware designs, either to speed up the verification process, or as a way of handling infinite state space. One of the most common methods to construct or refine abstractions is *Craig interpolation* [15], a logical tool to extract concise explanations for the (bounded) unreachability of error locations or states. To ensure rapid convergence, model checkers rely on theorem provers to find suitable interpolants, or interpolants containing the right predicates,

This work was partly supported by the ANR grant ANR-11- BS02-001-01 ReacHard, the EU FP7 STREP CERTAINTY, and the Swedish Research Council.

Jérôme Leroux
Univ. Bordeaux, 351 cours de la Libération, Talence, France
Tel.: +33 5 40 00 35 09
E-mail: leroux@labri.fr

Philipp Rümmer
Uppsala University, Box 337, 75105 Uppsala, Sweden
Tel.: +46 18 4713156
E-mail: philipp.ruemmer@it.uu.se

Pavle Subotić
Uppsala University, Box 337, 75105 Uppsala, Sweden
E-mail: pavle.subotic@it.uu.se

in a generally infinite lattice of interpolants for any given interpolation problem. In the past, a number of techniques have been proposed to guide theorem provers towards good interpolants (see Sect. 2 for an overview); however, those techniques either suffer from the fact that they require invasive changes to the theorem prover, or from the fact that they operate on a single proof of path infeasibility, and are therefore limited in the range of interpolants that can be produced.

We present a *semantic* framework for systematically exploring interpolant lattices, based on the notion of *interpolation abstraction.* Our approach is *solver-independent* and works by instrumenting the interpolation query, and therefore does not require any changes to the theorem prover. Despite simple implementation, interpolation abstractions are extremely flexible, and can incorporate domain-specific knowledge about promising interpolants, for instance in the form of *interpolant templates* used by the theorem prover. The framework can be used for a variety of logics, including arithmetic domains or programs operating on arrays or heap, and is also applicable for quantified interpolants.

We have integrated interpolation abstraction into the model checker Eldarica [33], which uses recursion-free Horn clauses (a generalisation of Craig interpolation) to construct abstractions [26,50]. Our experiments show that interpolation abstraction can prevent divergence of the model checker in cases that are often considered challenging.

On this article. This article is partly based on a paper at FMCAD 2013 [51]. Compared to the conference paper, the article considers a more general form of interpolation abstractions, with the result that central sections of the article were rewritten, as well as an extended experimental evaluation. New results also include Sect. 5.2, 6, 7.3, and 9.

1.1 Introductory Example

We consider an example inspired by the program discussed in the introduction of [36]. The example exhibits a situation that is generally considered challenging for automatic verifiers:

```
i = 0; x = j;
while (i<50) {i++; x++;}
if (j == 0) assert (x >= 50);
```

To show that the assertion holds, a predicate abstraction-based model checker would construct a set of inductive invariants as Boolean combination of given predicates. If needed, Craig interpolation is used to synthesise further predicates.

In the example, we might consider the path to the assertion in which the loop terminates after one iteration. This path could lead to an assertion violation if the conjunction of assignments and guards on the path (in SSA form) is satisfiable:

$$i_0 \doteq 0 \wedge x_0 \doteq j \wedge i_0 < 50 \wedge i_1 \doteq i_0 + 1 \wedge x_1 \doteq x_0 + 1 \tag{1}$$

$$\wedge\, i_1 \geq 50 \wedge j \doteq 0 \wedge x_1 < 50 \tag{2}$$

It is easy to see that the formula is unsatisfiable, and that the path therefore cannot cause errors. To obtain predicates that prevent the path from being considered again in the model checking process, Craig interpolants are computed for different partitionings of the conjuncts; we consider the case (1) ∧ (2), corresponding to the point on the path where the loop condition is checked for the second time. An interpolant is a formula I that satisfies the implications (1) → I and (2) → ¬I, and that only contains variables that occur in both (1) and (2); a model checker will use I as a candidate loop invariant.

The interpolation problem (1) ∧ (2) has several solutions, including $I_1 = (i_1 \leq 1)$ and $I_2 = (x_1 \geq i_1 + j)$. What makes the example challenging is the fact that a theorem prover is likely to compute interpolants like I_1, recognising the fact that the loop cannot terminate after only one iteration as obvious cause of infeasibility. I_1 does not describe a property that holds across loop iterations, however; after adding I_1 as a predicate, a model checker would have to consider the case that the loop terminates after two iterations, leading to a similar formula $i_2 \leq 2$, and so on. Model checking will only terminate after 50 loop unwindings; in similar situations with unbounded loops, picking interpolants like I_1 will lead to divergence (non-termination) of a model checker.

In contrast, the interpolant I_2 encodes a deeper explanation for infeasibility, the dependency between `i` and `x`, and takes the actual assertion to be verified into account. Since I_2 represents an inductive loop invariant, adding it as predicate will lead to significantly faster convergence of a model checker.

This article presents a methodology to systematically explore solutions of interpolation problems, enabling a model checker to steer the theorem prover towards interpolants like I_2. This is done by modifying the query given to the theorem prover, through the application of *interpolation abstractions* that capture domain knowledge about useful interpolants. To obtain I_2, we over-approximate the interpolation query (1) ∧ (2) in such a way that I_1 no longer is a valid interpolant:

$$(i_0 \doteq 0 \wedge x_0 \doteq j' \wedge i_0 < 50 \wedge i_1' \doteq i_0{+}1 \wedge x_1' \doteq x_0{+}1 \wedge \; x_1' - i_1' \doteq x_1 - i_1 \wedge j' \doteq j \;)$$
$$\wedge \; (\; x_1 - i_1 \doteq x_1'' - i_1'' \wedge j \doteq j'' \;\wedge i_1'' \geq 50 \wedge j'' \doteq 0 \wedge x_1'' < 50)$$

The rewriting consists of two parts: (i) the variables x_1, i_1, j are renamed to x_1', i_1', j' and x_1'', i_1'', j'', respectively; (ii) limited knowledge about the values of x_1, i_1, j is re-introduced, by adding the grey parts of the interpolation query. Note that the formula is still unsatisfiable. Intuitively, the theorem prover "forgets" the precise value of x_1, i_1, ruling out interpolants like I_1; however, the prover retains knowledge about the difference $x_1 - i_1$ (and the value of j), which is sufficient to compute relational interpolants like I_2.

The terms $x_1 - i_1$ and j have the role of *templates,* and encode the domain knowledge that linear relationships between variables and the loop counter are promising building blocks for invariants (the experiments in Sect. 8 illustrate the generality of this simple kind of template). Template-generated abstractions represent the most important class of interpolation abstractions considered in this article (but not the only one), and are extremely flexible: it is possible to use both template terms and template formulae, but also templates with quantifiers, parameters, or infinite sets of templates.

Templates are in our approach interpreted *semantically*, not *syntactically*, and it is up to the theorem prover to construct interpolants from templates, Boolean connectives, or other interpreted operations. For instance, the templates $x_1 - i_1$ and i_1 generate *the same* interpolation abstraction as the templates x_1 and i_1, since the values of the terms $x_1 - i_1, i_1$ uniquely determine the values of x_1, i_1, and vice versa. This adds to the flexibility of the framework, among others by including the possibility of disjunctive predicates at no extra cost.

1.2 Contributions and Organisation of the Article

- The framework of *interpolant abstractions* (Sect. 4);

- A catalogue of interpolation abstractions, in particular interpolation abstractions generated from *template terms* and *template predicates* (Sect. 5);
- *Algebraic properties* of the full space of interpolation abstractions (Sect. 6);
- Algorithms to explore *lattices of interpolation abstractions,* in order to compute a range of interpolants for a given interpolation problem (Sect. 7);
- An *experimental evaluation* using C programs and Horn clause benchmarks (Sect. 8);
- A complete case study on the use of interpolation abstractions for constructing effective algorithms for reachability analysis in *unbounded Petri nets* (Sect. 9).

2 Related Work

Syntactic restrictions of considered interpolants [36,44], for instance limiting the magnitude of literal constants in interpolants, can be used to enforce convergence and completeness of model checkers. This method is theoretically appealing, and has been the main inspiration for the work presented in this article. In practice, syntactic restrictions tend to be difficult to implement, since they require deep modifications of an interpolating theorem prover; in addition, completeness does not guarantee convergence within an acceptable amount of time. We present an approach that is semantic and more pragmatic in nature; while not providing any theoretic convergence guarantees, the use of domain-specific knowledge can lead to performance advantages in practice.

It has been proposed to use **term abstraction** to improve the quality of interpolants [2, 57]: intuitively, the occurrence of individual symbols in an interpolant can be prevented through renaming. Our approach is highly related to this technique, but is more general since it enables fine-grained control over symbol occurrences in an interpolant. For instance, in Sect. 1.1 arbitrary occurrence of the variable i_1 is forbidden, but occurrence in the context $x_1 - i_1$ is allowed.

The **strength of interpolants** can be controlled by choosing different interpolation calculi [18,48], applied to the same propositional resolution proof. To the best of our knowledge, there are few conclusive experiments relating interpolant strength with model checking performance. In [47], slight performance improvements are reported when using *weak* interpolants in the context of bounded model checking with function summaries; however, the results are dominated by improvements achieved when optimising the *size* of interpolants, preferring smaller over bigger interpolants. In addition, the extraction of different interpolants *from the same proof* is less flexible than imposing conditions already on the level of proof construction; if a proof does not leverage the right arguments why a program path is infeasible, it is unlikely that good interpolants can be extracted using any method.

Minimisation of proofs and interpolants through proof transformation [46,31,47] can have a positive impact on model checking performance; however, this is mainly due to the reduced overhead when processing smaller formulae, less due to a reduction in the number of refinement steps needed. The same comments as in the previous paragraph apply.

Divergence of model checkers can be prevented by combining interpolation with **acceleration**, which computes precise loop summaries for restricted classes of programs [12,53, 32]. Again, our approach is more pragmatic, can incorporate domain knowledge, but is not restricted to any particular class of programs. Our experiments show that our method is similarly effective as acceleration for preventing divergence when verifying error-free systems (Sect. 9). However, in contrast to acceleration, our method does not support the construction of long counterexamples spanning many loop iterations.

Templates have been used to synthesise program invariants in various contexts, for instance [21,4,55], and typically search for invariants within a rigidly defined set of constraints (e.g., with predefined Boolean or quantifier structure). Our approach can be used similarly, with complex building blocks for invariants specified by the user, but leaves the construction of interpolants from templates entirely to the theorem prover.

A number of systems compute interpolants by means of **constraint-based interpolation**, including CLP-Prover [52] and CSIsat [7]. This approach is similar in spirit to the template methods discussed in the previous paragraph, and imposes strong restrictions on the shape of considered interpolants. To the best of our knowledge, no attempts have been made to exploit domain-specific knowledge to guide constraint-based interpolation tools. Since our abstraction techniques are agnostic to the underlying interpolation engine, they can also be used in the context of constraint-based interpolation.

A recent paper proposes the generation of **beautiful interpolants** [1], which are interpolants with particularly simple shape and Boolean structure; empirically, interpolants of this kind were found to be beneficial for the convergence of model checkers. Domain-specific knowledge is not explicitly used when computing beautiful interpolants, but it is possible to use the procedure in [1] in combination with our abstraction framework.

Our interpolation abstraction technique has some similarities with the **implicit predicate abstraction** approach [56,14], which can be used to integrate abstraction into various model checking algorithms, including k-induction and IC3. Both in implicit predicate abstraction and in our approach, logical characterisations of abstractions are generated and handed over to a theorem prover. Implicit predicate abstraction has no direct relationship to interpolation, however.

Petri Nets are one of the most popular formal models for the representation and the analysis of parallel processes [19]. The reachability problem for Petri nets is central since many computational problems (even outside the realm of parallel processes) reduce to this problem. In 1981, Mayr [42] provided a proof of decidability of the reachability problem. Unfortunately, nowadays no tool implements a decision procedure for these problem. In fact, the algorithms provided by Mayr, and later by Kosaraju [37] and Lambert [38] are difficult to implement and computationally very expensive.

Recently [39,40], the reachability problem for Petri nets was proved to be decidable thanks to **inductive invariants in the Presburger arithmetic** (first-order logic over the integers with the addition). More precisely, if a marking $\bar{y}$ is not reachable from a marking $\bar{x}$, there exists an inductive invariant definable in Presburger arithmetic that contains the initial marking $\bar{x}$ but not the final one $\bar{y}$. Since we can decide if a Presburger formula denotes a forward inductive invariant, we deduce that there exist checkable certificates of non-reachability in the Presburger arithmetic. In particular, there exists a simple algorithm for deciding the general reachability problem based on two semi-algorithms. A first one that tries to prove the reachability by enumerating finite sequences of actions and a second one that tries to prove the non-reachability by enumerating Presburger formulas. Computing a sequence of actions leading the final marking, or a Presburger formula denoting a witness of non-reachability with efficient algorithms are two challenging problems; the application of interpolation abstractions to the latter is discussed in Sect. 9.

3 Preliminaries

3.1 Craig Interpolation

We assume familiarity with standard classical logic, including notions like terms, formulae, Boolean connectives, quantifiers, satisfiability, structures, models. For an overview, see, e.g., [27]. The main logics considered in this article are *classical first-order logic* with equality (FOL) and *Presburger arithmetic* (PA), but our method is not restricted to FOL or PA. In the context of SMT, the quantifier-free fragment of FOL, with equality $\doteq$ as the only predicate, is usually denoted by EUF.

Given any logic, we distinguish between *logical symbols,* which include Boolean connectives, equality $\doteq$, interpreted functions, etc., and *non-logical symbols*, among others variables and uninterpreted functions. If $\bar{s} = \langle s_1, \ldots, s_n \rangle$ is a list of non-logical symbols, we write $\phi[\bar{s}]$ (resp., $t[\bar{s}]$) for a formula (resp., term) containing no non-logical symbols other than $\bar{s}$. We write $\bar{s}' = \langle s'_1, \ldots, s'_n \rangle$ (and similarly $\bar{s}''$, etc.) for a list of primed symbols; $\phi[\bar{s}']$ (resp., $t[\bar{s}']$) is the variant of $\phi[\bar{s}]$ (resp., $t[\bar{s}]$) in which $\bar{s}$ has been replaced with $\bar{s}'$.

An interpolation problem is a conjunction $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ over disjoint lists $\bar{s}_A, \bar{s}, \bar{s}_B$ of symbols. An *interpolant* is a formula $I[\bar{s}]$ such that $A[\bar{s}_A, \bar{s}] \Rightarrow I[\bar{s}]$ and $B[\bar{s}, \bar{s}_B] \Rightarrow \neg I[\bar{s}]$. We say that an interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is *solvable* if an interpolant exists; solvability of the problem implies that $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is unsatisfiable. We say that a logic has the *interpolation property* if also the opposite holds: whenever $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is unsatisfiable, there is an interpolant $I[\bar{s}]$. For sake of presentation, we only consider logics with the interpolation property.[1]

We represent *binary relations* as formulae $R[\bar{s}_1, \bar{s}_2]$ over two lists $\bar{s}_1, \bar{s}_2$ of symbols, and relations over a vocabulary $\bar{s}$ as $R[\bar{s}, \bar{s}']$. The identity relation over $\bar{s}$ is denoted by $Id[\bar{s}, \bar{s}']$.

With slight abuse of notation, if $\phi[x_1, \ldots, x_n]$ is a formula containing the free variables $x_1, \ldots, x_n$, and $t_1, \ldots, t_n$ are ground terms, then we write $\phi[t_1, \ldots, t_n]$ for the formula obtained by substituting $t_1, \ldots, t_n$ for $x_1, \ldots, x_n$.

3.2 Stateless Logics

Some of the results presented in this article require an additional assumption about a logic:

Definition 1 A logic is called *stateless* if conjunctions $A[\bar{s}] \wedge B[\bar{t}]$ of satisfiable formulae $A[\bar{s}]$, $B[\bar{t}]$ over disjoint lists $\bar{s}, \bar{t}$ of non-logical symbols are satisfiable.

Intuitively, formulae in a stateless logic interact only through non-logical symbols, not via any notion of global state, structure, etc. Many logics that are relevant in the context of verification are stateless, in particular quantifier-free FOL, PA, logics based on the theory of arrays, etc. An example of a stateful logic is full FOL with equality. For instance, consider the conjunction $(\forall x, y.\ x \doteq y) \wedge (\exists x, y.\ x \neq y)$ in full FOL. Although the individual conjuncts $\forall x, y.\ x \doteq y$ and $\exists x, y.\ x \neq y$ are satisfiable, their conjunction is not: the first conjunct enforces a universe with only one element, whereas the second conjunct requires at least two elements.

[1] The concept of interpolation abstraction can in principle also be used with other logics (for instance, the theory of arrays without quantifiers, most versions of which do not have the interpolation property), at the cost of modifying some of the algorithms in the article. Among others, in Sect. 7 it would no longer be possible to check existence of interpolants by just proving unsatisfiability of constructed formulae.

Other stateful logics are modal logic or separation logic; often, such logics can naturally be made stateless by enriching their vocabulary. Statelessness is important in this article, since we use the concept of *renaming* of symbols to ensure independence of formulae.

3.3 Orders and Lattices

A *poset* is a set D equipped with a partial ordering $\sqsubseteq$. A poset $\langle D, \sqsubseteq\rangle$ is *bounded* if it has a *least element* $\bot$ and a *greatest element* $\top$. We denote the *least upper bound* and the *greatest lower bound* of a set $X \subseteq D$ by $\bigsqcup X$ and $\bigsqcap X$, respectively, provided that they exist. Given elements $a, b \in D$, we say b is a *successor* (resp., *predecessor*) of a if $a \sqsubseteq b$ but $a \neq b$, and *immediate successor* if in addition there is no $c \in D \setminus \{a, b\}$ with $a \sqsubseteq c \sqsubseteq b$ (resp., *immediate predecessor*). Elements $a, b \in D$ with $a \not\sqsubseteq b$ and $b \not\sqsubseteq a$ are *incomparable*. An element $a \in X \subseteq D$ is a *maximal element* (resp., *minimal element*) of X if $a \sqsubseteq b$ (resp., $b \sqsubseteq a$) and $b \in X$ imply $a = b$.

A *lattice* $L = \langle D, \sqsubseteq\rangle$ is a poset $\langle D, \sqsubseteq\rangle$ such that $a \sqcup b = \bigsqcup\{a, b\}$ and $a \sqcap b = \bigsqcap\{a, b\}$ exist for all $a, b \in D$. L is a *complete lattice* if all non-empty subsets $X \subseteq D$ have a least upper bound and greatest lower bound. A complete lattice is bounded by definition. A non-empty subset $M \subseteq D$ forms a *sub-lattice* if $a \sqcup b \in M$ and $a \sqcap b \in M$ for all $a, b \in M$. A sub-lattice $M \subseteq D$ is *convex* if $a \sqsubseteq c \sqsubseteq b$ and $a, b \in M$ imply $c \in M$. A lattice is *distributive* if for all $a, b, c \in D$, $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$. A *completely distributive lattice* is a complete lattice in which arbitrary joins ($\sqcup$) distribute over arbitrary meets ($\sqcap$).

A function $f : D_1 \to D_2$, where $\langle D_1, \sqsubseteq_1\rangle$ and $\langle D_2, \sqsubseteq_2\rangle$ are posets, is *monotonic* if $x \sqsubseteq_1 y$ implies $f(x) \sqsubseteq_2 f(y)$, and *anti-monotonic* if $x \sqsubseteq_1 y$ implies $f(y) \sqsubseteq_2 f(x)$.

4 Interpolation Abstractions

4.1 Basic Definitions

This section defines the concept of interpolation abstractions, and derives basic properties. Interpolation abstractions are represented by transformations of the formulae to be interpolated; in the most general formulation, this is represented via a pair of extensive functions on formulae:

Definition 2 (Interpolation abstraction) Suppose $\bar{s}$ is a list of non-logical symbols, for some arbitrary but fixed logic. An *interpolation abstraction* is a pair (T_A, T_B) of functions mapping formulae to formulae, with the following properties:

(i) for any formula $A[\bar{s}_A, \bar{s}]$, and $\bar{s}_A$ disjoint from $\bar{s}$, the result of applying T_A is a new formula $A'[\bar{s}_{A'}, \bar{s}] = T_A(A[\bar{s}_A, \bar{s}])$ (again with $\bar{s}_{A'}$ and $\bar{s}$ disjoint) such that the implication $(\exists \bar{s}_A.\, A[\bar{s}_A, \bar{s}]) \Rightarrow (\exists \bar{s}_{A'}.\, A'[\bar{s}_{A'}, \bar{s}])$ holds.[2]

(ii) similarly, for any formula $B[\bar{s}, \bar{s}_B]$, and $B'[\bar{s}, \bar{s}_{B'}] = T_B(B[\bar{s}, \bar{s}_B])$, it is the case that $(\exists \bar{s}_B.\, B[\bar{s}, \bar{s}_B]) \Rightarrow (\exists \bar{s}_{B'}.\, B'[\bar{s}, \bar{s}_{B'}])$.

(iii) for any $A[\bar{s}_A, \bar{s}]$, $B[\bar{s}, \bar{s}_B]$, and disjoint lists $\bar{s}_A, \bar{s}, \bar{s}_B$ of symbols, the common symbols of the formulae $T_A(A[\bar{s}_A, \bar{s}])$ and $T_B(B[\bar{s}, \bar{s}_B])$ are contained in $\bar{s}$.

[2] Since $\bar{s}_A$, $\bar{s}_{A'}$ might contain functions or predicates, depending on the considered logic, the quantifiers can be higher order. An equivalent model-theoretic definition, avoiding the quantifiers, would be: "For every model S of $A[\bar{s}_A, \bar{s}]$, there is a model S' of $A'[\bar{s}_{A'}, \bar{s}]$ that agrees with S on the interpretation of $\bar{s}$."

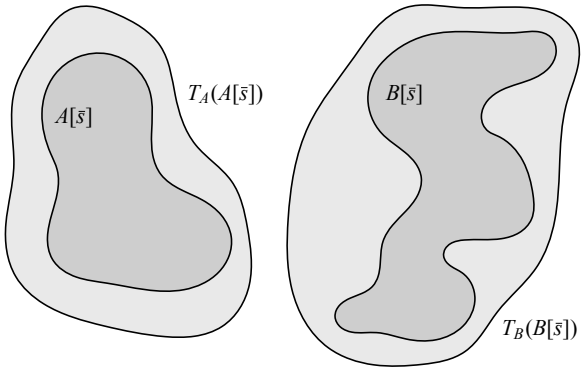


Fig. 1: Illustration of interpolation abstraction, assuming that only common non-logical symbols exist. Both the concrete and abstract problem are solvable.

We call $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ a *concrete interpolation problem,* and $T_A(A[\bar{s}_A, \bar{s}]) \wedge T_B(B[\bar{s}, \bar{s}_B])$ the corresponding *abstract interpolation problem* for the interpolation abstraction (T_A, T_B).

In other words, interpolation abstractions define over-approximations of the conjuncts to be interpolated. Assuming that the concrete interpolation problem is solvable, we call the interpolation abstraction *feasible* if also the abstract interpolation problem is solvable, and *infeasible* otherwise.

Example 1 An illustration is given in Fig. 1. The concrete interpolation problem is solvable since the solution sets $A[\bar{s}]$ and $B[\bar{s}]$ are disjoint, i.e., $A[\bar{s}] \wedge B[\bar{s}]$ is unsatisfiable. An interpolant is a formula $I[\bar{s}]$ that represents a superset of $A[\bar{s}]$, but that is disjoint with $B[\bar{s}]$. By definition, the formula $T_A(A[\bar{s}])$ represents an *over-approximation* of $A[\bar{s}]$; similarly for $T_B(B[\bar{s}])$. This ensures the soundness of computed abstract interpolants (see Lem. 1 below). In Fig. 1, despite over-approximation, the abstract interpolation problem is solvable, which means that the interpolation abstraction is feasible.

While there are many ways to construct interpolation abstractions, in the scope of this article we mainly concentrate on interpolation abstractions defined by means of *relations:*

Definition 3 (Relation abstraction) Suppose $\bar{s}$ is a list of non-logical symbols, and $\bar{s}'$ and $\bar{s}''$ fresh copies of $\bar{s}$. A *relation abstraction* is a pair $(R_A[\bar{s}', \bar{s}], R_B[\bar{s}, \bar{s}''])$ of formulae with the property that $R_A[\bar{s}, \bar{s}]$ and $R_B[\bar{s}, \bar{s}]$ are valid (i.e., $Id[\bar{s}', \bar{s}] \Rightarrow R_A[\bar{s}', \bar{s}]$ and $Id[\bar{s}, \bar{s}''] \Rightarrow R_B[\bar{s}, \bar{s}'']$). A relation abstraction defines an interpolation abstraction (T_A, T_B) by:

$$T_A(A[\bar{s}_A, \bar{s}]) \;=\; A[\bar{s}_A, \bar{s}'] \wedge R_A[\bar{s}', \bar{s}], \qquad T_B(B[\bar{s}, \bar{s}_B]) \;=\; R_B[\bar{s}, \bar{s}''] \wedge B[\bar{s}'', \bar{s}_B]\,.$$

Thus, the relation abstraction of a concrete interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is

$$(A[\bar{s}_A, \bar{s}'] \wedge R_A[\bar{s}', \bar{s}]) \;\wedge\; (R_B[\bar{s}, \bar{s}''] \wedge B[\bar{s}'', \bar{s}_B])\,.$$

Note that properties (i) and (ii) in Def. 2 are ensured by requiring that the relations $R_A[\bar{s}', \bar{s}]$ and $R_B[\bar{s}, \bar{s}'']$ subsume the identity relation ($R_A[\bar{s}, \bar{s}]$ and $R_B[\bar{s}, \bar{s}]$ are valid).

Example 2 The interpolation abstraction applied in Sect. 1.1 is a relation abstraction. The common symbols of the interpolation problem are $\bar{s} = \langle x_1, i_1, j \rangle$, and the relation abstraction is defined by $R_A = (x_1' - i_1' \doteq x_1 - i_1 \wedge j' \doteq j)$ and $R_B = (x_1 - i_1 \doteq x_1'' - i_1'' \wedge j \doteq j'')$.

Finally, we can state a (straightforward) result about the correctness of interpolants computed using interpolation abstractions:

Lemma 1 (Soundness) *Every interpolant of the abstract interpolation problem is also an interpolant of the concrete interpolation problem (but in general not vice versa).*

Proof Suppose $A'[\bar{s}_{A'}, \bar{s}] = T_A(A[\bar{s}_A, \bar{s}])$ and $B'[\bar{s}, \bar{s}_{B'}] = T_B(B[\bar{s}, \bar{s}_B])$. An abstract interpolant only contains symbols from $\bar{s}$ (due to property (iii) of Def. 2), i.e., is of the form $I[\bar{s}]$. It also satisfies $A'[\bar{s}_{A'}, \bar{s}] \Rightarrow I[\bar{s}]$ and $B'[\bar{s}, \bar{s}_{B'}] \Rightarrow \neg I[\bar{s}]$, and thus $\exists \bar{s}_{A'}.\, A'[\bar{s}_{A'}, \bar{s}] \Rightarrow I[\bar{s}]$ and $\exists \bar{s}_{B'}.\, B'[\bar{s}, \bar{s}_{B'}] \Rightarrow \neg I[\bar{s}]$. Thanks to properties (i) and (ii) in Def. 2, this yields the implications $A[\bar{s}_A, \bar{s}] \Rightarrow I[\bar{s}]$ and $B[\bar{s}, \bar{s}_B] \Rightarrow \neg I[\bar{s}]$. □

4.2 Interpolant Lattices

Interpolation abstractions can be used to guide interpolation engines, by restricting the space $\mathit{Inter}(A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B])$ of interpolants satisfying an interpolation problem. Since the set $\mathit{Inter}(A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B])/\equiv$ of interpolant classes (modulo logical equivalence) is closed under conjunctions and disjunctions, the structure $(\mathit{Inter}(A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B])/\equiv, \Rightarrow)$ forms a lattice. Fig. 2 shows the interpolant lattice for the example in Sect. 1.1; this lattice has a strongest concrete interpolant $I_\bot$ and a weakest concrete interpolant $I_\top$.[3]

For a feasible abstraction, the lattice $(\mathit{Inter}(T_A(A[\bar{s}_A, \bar{s}']),\ T_B(B[\bar{s}'', \bar{s}_B]))/\equiv,\ \Rightarrow\ )$ of abstract interpolants is a sub-lattice of the concrete interpolant lattice. The sub-lattice is *convex,* because if I_1 and I_3 are abstract interpolants and I_2 is a concrete interpolant with $I_1 \Rightarrow I_2 \Rightarrow I_3$, then also I_2 is an abstract interpolant. The choice of the function T_A in an interpolation abstraction constrains the lattice of abstract interpolants from below, the function T_B from above.

We illustrate two disjoint sub-lattices in Fig. 2: the left box is the sub-lattice for the relation abstraction $(i_1' \doteq i_1, i_1 \doteq i_1'')$, while the right box represents the relation abstraction

$$(x_1' - i_1' \doteq x_1 - i_1 \wedge j' \doteq j,\ x_1 - i_1 \doteq x_1'' - i_1'' \wedge j \doteq j'')$$

used in Sect. 1.1 to derive interpolant I_2.

As the following lemma shows, there are no principal restrictions how fine-grained the guidance enforced by an interpolation abstraction can be; however, since abstraction is a semantic notion, we can only impose constraints *up to equivalence of interpolants:*

Lemma 2 (Completeness) *Suppose $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is an interpolation problem with interpolant $I[\bar{s}]$ in a stateless logic, such that both $A[\bar{s}_A, \bar{s}]$ and $B[\bar{s}, \bar{s}_B]$ are satisfiable (the problem is not degenerate). Then there is a feasible interpolation abstraction, definable as a relation abstraction in the same logic, such that every abstract interpolant is logically equivalent to $I[\bar{s}]$.*

[3] In general, the interpolant lattice might be incomplete and not contain such elements.

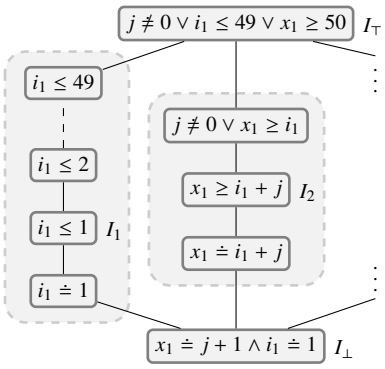


Fig. 2: Parts of the interpolant lattice for the example in Sect. 1.1 (up to equivalence). The dashed boxes represent the sub-lattices for the abstraction induced by the template terms $\{i_1\}$ (left) and $\{x_1 - i_1, j\}$ (right).

Proof Choose the relation abstraction $(I[\bar{s}'] \rightarrow I[\bar{s}],\ I[\bar{s}] \rightarrow I[\bar{s}''])$. Since $I[\bar{s}]$ is an interpolant of the abstract interpolation problem, the abstract problem is solvable. Further, assume that $I'[\bar{s}]$ is an arbitrary abstract interpolant, i.e.,

$$A[\bar{s}_A, \bar{s}'] \wedge (I[\bar{s}'] \rightarrow I[\bar{s}]) \;\Rightarrow\; I'[\bar{s}] \quad \text{and} \quad (I[\bar{s}] \rightarrow I[\bar{s}'']) \wedge B[\bar{s}'', \bar{s}_B] \;\Rightarrow\; \neg I'[\bar{s}]\,.$$

By rewriting the left-hand sides of the entailments, we can conclude $I[\bar{s}] \Leftrightarrow I'[\bar{s}]$. We only show one of the directions:

$$\begin{aligned} A[\bar{s}_A, \bar{s}'] \wedge (I[\bar{s}'] \rightarrow I[\bar{s}]) \;&\Leftrightarrow\; (A[\bar{s}_A, \bar{s}'] \wedge \neg I[\bar{s}']) \vee (A[\bar{s}_A, \bar{s}'] \wedge I[\bar{s}]) \\ &\Leftrightarrow\; A[\bar{s}_A, \bar{s}'] \wedge I[\bar{s}] \end{aligned}$$

From $(A[\bar{s}_A, \bar{s}'] \wedge I[\bar{s}]) \Rightarrow I'[\bar{s}]$, it follows that $I[\bar{s}] \Rightarrow I'[\bar{s}]$, since $A[\bar{s}_A, \bar{s}']$ is satisfiable and does not contain any symbols from $\bar{s}$, and the considered logic is stateless. □

5 A Catalogue of Interpolation Abstractions

This section introduces a range of practically relevant relation abstractions, mainly defined in terms of *templates* as illustrated in Sect. 1.1. For any interpolation abstraction, it is interesting to consider the following questions:

(i) provided the concrete interpolation problem is solvable, characterise the cases in which also the abstract problem can be solved (how *coarse* the abstraction is);
(ii) provided the abstract interpolation problem is solvable, characterise the space of abstract interpolants.

The first point touches the question to which degree an interpolation abstraction limits the set of proofs that a theorem prover can find. We hypothesise (and explain in Sect. 1.1) that it is less important to generate interpolants with a specific syntactic shape, than to force a theorem prover to use the *right argument* for showing that a path in a program is safe.

We remark that interpolation abstractions can also be combined, for instance to create abstractions that include both template terms and template predicates. In general, the component-wise conjunction of two interpolation abstractions is again a well-formed abstraction, as is the disjunction. More details are given in Sect. 6.

5.1 Finite Term Interpolation Abstractions

The first family of interpolation abstractions is defined with the help of finite sets T of *template terms,* and formalises the abstraction used in Sect. 1.1. Intuitively, abstract interpolants for a term abstraction induced by T are formulae that only use elements of T, in combination with logical symbols, as building blocks (a precise characterisation is given in Lem. 4 below). For the case of interpolation in EUF (quantifier-free FOL without uninterpreted predicates), this means that abstract interpolants are Boolean combinations of equations between T terms. In linear arithmetic, abstract interpolants may contain equations and inequalities over linear combinations of T terms.

The relations defining a term interpolation abstraction follow the example given in Sect. 1.1, and assert that primed and unprimed versions of T terms have the same value. As a consequence, nothing is known about the value of unprimed terms that are *not* mentioned in T.

Definition 4 (Term interpolation abstraction) Suppose $\bar{s}$ is a list of non-logical symbols, $\bar{s}'$ and $\bar{s}''$ fresh copies of $\bar{s}$, and $T = \{t_1[\bar{s}], \ldots, t_n[\bar{s}]\}$ a finite set of ground terms. The relation abstraction $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ defined by

$$R_A^T[\bar{s}', \bar{s}] \;=\; \bigwedge_{i=1}^{n} t_i[\bar{s}'] \doteq t_i[\bar{s}], \qquad R_B^T[\bar{s}, \bar{s}''] \;=\; \bigwedge_{i=1}^{n} t_i[\bar{s}] \doteq t_i[\bar{s}'']$$

is called *term interpolation abstraction* over T.

Term abstractions are feasible if and only if a concrete interpolant exists that can be expressed purely using T terms:

Lemma 3 (Solvability) *Suppose $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is an interpolation problem, and $T = \{t_1[\bar{s}], \ldots, t_n[\bar{s}]\}$ a finite set of ground terms. The abstract interpolation problem for the abstraction $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ is solvable if and only if there is a formula $I[x_1, \ldots, x_n]$ over n variables $x_1, \ldots, x_n$ (and no further non-logical symbols) such that $I[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is an interpolant of $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$.*

Proof "⇐": $I[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is also an abstract interpolant, which implies that the abstract interpolation problem is solvable.

"⇒": suppose the abstract interpolation problem

$$\left(A[\bar{s}_A, \bar{s}'] \wedge \bigwedge_{i=1}^{n} t_i[\bar{s}'] \doteq t_i[\bar{s}]\right) \;\wedge\; \left(\bigwedge_{i=1}^{n} t_i[\bar{s}] \doteq t_i[\bar{s}''] \wedge B[\bar{s}'', \bar{s}_B]\right) \tag{3}$$

is solvable, which means that (3) is an unsatisfiable formula. Then also the following formula is unsatisfiable (for fresh variables $x_1, \ldots, x_n$):

$$\left(A[\bar{s}_A, \bar{s}'] \wedge \bigwedge_{i=1}^{n} t_i[\bar{s}'] \doteq x_i\right) \;\wedge\; \left(\bigwedge_{i=1}^{n} x_i \doteq t_i[\bar{s}''] \wedge B[\bar{s}'', \bar{s}_B]\right) \tag{4}$$

Namely, suppose (4) is satisfied by the model S. The model can be extended to a model S' of (3) by interpreting the symbols $\bar{s}$ with the same value as the symbols $\bar{s}'$.

Given that (4) is unsatisfiable, due to the interpolation property there is an interpolant $I[x_1, \ldots, x_n]$ for (4). By the substitution theorem, then also $I[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is an interpolant for (3). Finally, by Lem. 1, $I[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is also an interpolant of the original interpolant problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$. □

Example 3 Consider the interpolation abstraction used in Sect. 1.1, which is created by the set $T = \{x_1 - i_1, j\}$ of terms. The abstract interpolation problem is solvable with interpolant $x_1 \geq i_1 + j$, which can be represented as $(x_1 - i_1) \geq (j)$ as a combination of the template terms in T.

It would be tempting to assume that *all* interpolants generated by term interpolation abstractions are as specified in Lem. 3, i.e., constructed only from T terms and logical symbols. In fact, since our framework restricts the space of interpolants in a semantic way, only weaker guarantees can be provided about the range of possible interpolants; this is related to the earlier observation (Sect. 4) that interpolation can only be restricted *up to logical equivalence:*

Lemma 4 (Interpolant space) *Suppose the abstract interpolation problem for the relation abstraction* $(R^T_A[\bar{s}', \bar{s}], R^T_B[\bar{s}, \bar{s}''])$ *is solvable, and the underlying logic is EUF or PA. Then there is a strongest abstract interpolant* $I_\bot[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$*, and a weakest abstract interpolant* $I_\top[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$*, each constructed only from* T *terms and logical symbols. A formula* $J[\bar{s}]$ *is an abstract interpolant iff the implications* $I_\bot[t_1[\bar{s}], \ldots, t_n[\bar{s}]] \Rightarrow J[\bar{s}] \Rightarrow I_\top[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ *hold.*

Proof Again consider the interpolation problem (4), and observe that there is a strongest interpolant $I_\bot[x_1, \ldots, x_n]$ and a weakest interpolant $I_\top[x_1, \ldots, x_n]$. (For EUF, this is because there are only finitely many interpolants up to equivalence; for PA, this holds due to the quantifier elimination property).

We show that $I_\bot[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is the conjectured strongest interpolant. (The proof for the weakest interpolant $I_\top[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is symmetric.) Suppose $J[\bar{s}]$ is any abstract interpolant, which means

$$\left(A[\bar{s}_A, \bar{s}'] \wedge \bigwedge_{i=1}^{n} t_i[\bar{s}'] \doteq t_i[\bar{s}]\right) \;\Rightarrow\; J[\bar{s}]$$

and therefore also

$$\left(A[\bar{s}_A, \bar{s}'] \wedge \bigwedge_{i=1}^{n} t_i[\bar{s}'] \doteq x_i\right) \;\Rightarrow\; \left(\bigwedge_{i=1}^{n} x_i \doteq t_i[\bar{s}] \;\Rightarrow\; J[\bar{s}]\right)$$

Since left-hand and right-hand side only share the (uninterpreted) symbols $x_1, \ldots, x_n$, and $I_\bot[x_1, \ldots, x_n]$ is the strongest formula over those symbols implied by the left-hand side, this entails:

$$I_\bot[x_1, \ldots, x_n] \;\Rightarrow\; \left(\bigwedge_{i=1}^{n} x_i \doteq t_i[\bar{s}] \;\Rightarrow\; J[\bar{s}]\right)$$

and therefore $I_\bot[t_1[\bar{s}], \ldots, t_n[\bar{s}]] \Rightarrow J[\bar{s}]$. □

Example 4 Again, consider Sect. 1.1, and the interpolant lattice as shown in Fig. 2. The strongest abstract interpolant for the interpolation abstraction induced by $T = \{x_1 - i_1, j\}$ is $x_1 \doteq i_1 + j$, the weakest one $j \neq 0 \vee x_1 \geq i_1$.

In the case of a logic with arithmetic operators, for instance linear rational arithmetic or Presburger arithmetic, it is possible to define interpolation abstractions on the basis of inequalities instead of equations, to achieve more fine-grained control over interpolants. Inequality interpolation abstractions can specify that interpolants can only give upper bounds (or only lower bounds) on the value of some term t, i.e., t can only occur on the left- or right-hand side of inequalities $\leq$, and not as part of equations. This degree of control is useful for model checking applications, where it is well-known that the quality of interpolants can be improved by abstracting equations to inequalities.

Definition 5 (Inequality interpolation abstraction) Suppose $\bar{s}$ is a list of non-logical symbols, $\bar{s}'$ and $\bar{s}''$ fresh copies of $\bar{s}$, and $T = \{t_1[\bar{s}], \ldots, t_n[\bar{s}]\}$ a finite set of ground terms. The relation abstraction $(R_A^{\leq T}[\bar{s}', \bar{s}], R_B^{\leq T}[\bar{s}, \bar{s}''])$ defined by

$$R_A^{\leq T}[\bar{s}', \bar{s}] \;=\; \bigwedge_{i=1}^{n} t_i[\bar{s}'] \leq t_i[\bar{s}], \qquad R_B^{\leq T}[\bar{s}, \bar{s}''] \;=\; \bigwedge_{i=1}^{n} t_i[\bar{s}] \leq t_i[\bar{s}'']$$

is called *inequality interpolation abstraction* over T.

Intuitively, the terms T can only occur only the right side of inequalities $\leq$ in interpolants, i.e., in the form of lower bounds. To specify upper bounds, it is possible to provide negative terms $-t \in T$; when including both t and $-t$ in T, arbitrary occurrences of t in an interpolant are possible (also within equations). This shows that inequality interpolation abstractions strictly subsume term interpolation abstractions in the presence of arithmetic.

To characterise solvability, assume that interpolants only contain inequalities $\leq$ (and no $\geq$ or equations $\doteq$), and that no inequalities occur underneath negation $\neg$. An occurrence of a term is then called *positive* if the term (or a positive multiple of the term) is on the right-hand side of $\leq$, and negative if it is on the left-hand side.

Lemma 5 (Solvability) *Suppose $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is an interpolation problem in PA, and $T = \{t_1[\bar{s}], \ldots, t_n[\bar{s}]\}$ a finite set of ground terms. The abstract interpolation problem for the abstraction $(R_A^{\leq T}[\bar{s}', \bar{s}], R_B^{\leq T}[\bar{s}, \bar{s}''])$ is solvable if and only if there is a formula $I[x_1, \ldots, x_n]$ over n variables $x_1, \ldots, x_n$, all occurring only positively in $I[x_1, \ldots, x_n]$, such that the formula $I[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is an interpolant of $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$.*

Proof "$\Leftarrow$": as for Lem. 3, it can be observed that $I[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is also an abstract interpolant, which implies that the abstract interpolation problem is solvable.

"$\Rightarrow$": again, as for Lem. 3, we consider the modified interpolation problem

$$\left(A[\bar{s}_A, \bar{s}'] \wedge \bigwedge_{i=1}^{n} t_i[\bar{s}'] \leq x_i\right) \;\wedge\; \left(\bigwedge_{i=1}^{n} x_i \leq t_i[\bar{s}''] \wedge B[\bar{s}'', \bar{s}_B]\right)$$

As a constructive way of showing the existence of interpolants $I[x_1, \ldots, x_n]$ of the desired form, the interpolating PA sequent calculus from [10] can be used. To this end, first assume that the conjuncts $A[\bar{s}_A, \bar{s}]$, $B[\bar{s}, \bar{s}_B]$ are normalised in such a way that no equations, no quantifiers (or divisibility constraints), and no negations occur. Then, construct an interpolating proof by strictly ordering the applied rules: (i) eliminate all Boolean operators, (ii) apply arithmetic rules to the literals obtained from $A[\bar{s}_A, \bar{s}']$, $B[\bar{s}'', \bar{s}_B]$, (iii) finally, resolve with the inequalities $t_i[\bar{s}'] \leq x_i$ and $x_i \leq t_i[\bar{s}'']$ to obtain conflicts and close proof goals.

By checking the individual proof rules in [10], we can observe that the resulting interpolant $I[x_1, \ldots, x_n]$ will only contain variables $x_1, \ldots, x_n$ in positive positions. □

In a similar way as sets of terms, also finite sets of *formulae* induce interpolation abstractions. Template formulae can be relevant to steer an interpolating theorem prover towards (possibly user-specified or quantified) interpolants that might be hard to find for the prover alone. The approach bears some similarities to the concept of predicate abstraction in model checking [25, 28], but still leaves the use of templates entirely to the theorem prover.

Definition 6 (Predicate interpolation abstraction) Suppose $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is an interpolation problem, and $Pred = \{\phi_1[\bar{s}], \ldots, \phi_n[\bar{s}]\}$ is a finite set of formulae. The relation abstraction $(R_A^{Pred}[\bar{s}', \bar{s}], R_B^{Pred}[\bar{s}, \bar{s}''])$ defined by

$$R_A^{Pred}[\bar{s}', \bar{s}] \;=\; \bigwedge_{i=1}^{n} (\phi_i[\bar{s}'] \rightarrow \phi_i[\bar{s}]), \qquad R_B^{Pred}[\bar{s}, \bar{s}''] \;=\; \bigwedge_{i=1}^{n} (\phi_i[\bar{s}] \rightarrow \phi_i[\bar{s}''])$$

is called *predicate interpolation abstraction* over *Pred*.

Intuitively, predicate interpolation abstractions restrict the solutions of an interpolation problem to those interpolants that can be represented as a positive Boolean combination of the predicates in *Pred* (i.e., by combining elements of *Pred* using $\wedge$ and $\vee$, without negations $\neg$). Note that it is possible to include the negation of a predicate $\phi[\bar{s}]$ in *Pred* if *negative* occurrences of $\phi[\bar{s}]$ are supposed to be allowed in an interpolant (or both $\phi[\bar{s}]$ and $\neg\phi[\bar{s}]$ for both positive and negative occurrences).

Lemma 6 (Solvability) *Suppose $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is an interpolation problem, and Pred a finite set of predicates. If the underlying logic is stateless, then the abstract interpolation problem for $(R_A^{Pred}[\bar{s}', \bar{s}], R_B^{Pred}[\bar{s}, \bar{s}''])$ is solvable if and only if $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ has an interpolant $I[\bar{s}]$ that is a positive Boolean combination of predicates in Pred.*

Proof "$\Leftarrow$": via Boolean reasoning, it can be shown that the interpolant $I[\bar{s}]$ also is a solution of the abstract problem $(A[\bar{s}_A, \bar{s}'] \wedge R_A^{Pred}[\bar{s}', \bar{s}]) \wedge (R_B^{Pred}[\bar{s}, \bar{s}''] \wedge B[\bar{s}'', \bar{s}_B])$.

"$\Rightarrow$": suppose $(A[\bar{s}_A, \bar{s}'] \wedge R_A^{Pred}[\bar{s}', \bar{s}]) \wedge (R_B^{Pred}[\bar{s}, \bar{s}''] \wedge B[\bar{s}'', \bar{s}_B])$ is unsatisfiable. As a constructive way to show the existence of an interpolant that is a positive Boolean combination of *Pred* predicates, we use the propositional interpolating calculus from [11, Fig. 1]. Thanks to proof-confluency, we can start by splitting all implications from $R_A^{Pred}[\bar{s}', \bar{s}]$ and $R_B^{Pred}[\bar{s}, \bar{s}'']$, using rules OR-LEFT, NOT-LEFT. After that, all sequents containing complementary formulae $\phi_i[\bar{s}]$ can be closed with the rule CLOSE-LR; this leads to positive occurrences of $\phi_i[\bar{s}]$ in the interpolant.

All other sequents have the form $\ldots, \lfloor A[\bar{s}_A, \bar{s}'] \rfloor_L, \lfloor B[\bar{s}'', \bar{s}_B] \rfloor_R, BP \;\vdash\; AP, \ldots$ where AP is a set of formulae of the form $\lfloor \phi_i[\bar{s}'] \rfloor_L$, and BP a set of formulae $\lfloor \phi_j[\bar{s}''] \rfloor_R$. Since the sequent is valid by assumption, and since the underlying logic is stateless, at least one of $A[\bar{s}_A, \bar{s}'] \wedge \neg AP$ and $B[\bar{s}'', \bar{s}_B] \wedge BP$ is unsatisfiable. In the first case, the sequent can be closed with interpolant *false*, in the latter case with interpolant *true*. □

We remark that the implication $\Leftarrow$ holds in all cases, whereas $\Rightarrow$ needs the assumption that the logic is stateless. As a counterexample for the stateful case, consider again the interpolation problem $(\forall x, y.\ x \doteq y) \wedge (\exists x, y.\ x \not\doteq y)$ in full FOL. The abstract interpolation problem is solvable even for $Pred = \emptyset$ (with interpolant $\forall x, y.\ x \doteq y$), but no positive Boolean combination of *Pred* formulae is an interpolant.

The interpolant space can be characterised as for term interpolation abstractions (Lem. 4):

Lemma 7 (Interpolant space) *Suppose the abstract interpolation problem for the relation abstraction* $(R_A^{Pred}[\bar{s}', \bar{s}], R_B^{Pred}[\bar{s}, \bar{s}''])$ *is solvable, and the underlying logic is stateless. Then there is a strongest abstract interpolant* $I_\bot[\bar{s}]$*, and a weakest abstract interpolant* $I_\top[\bar{s}]$*, each being a positive Boolean combination of predicates in Pred. A formula* $J[\bar{s}]$ *is an abstract interpolant iff the implications* $I_\bot[\bar{s}] \Rightarrow J[\bar{s}] \Rightarrow I_\top[\bar{s}]$ *hold.*

Proof As in the proof Lem. 4, but with Boolean variables instead of $x_1, \ldots, x_n$. □

5.4 Quantified Interpolation Abstractions

The previous sections showed how interpolation abstractions are generated by finite sets of templates. A similar construction can be performed for *infinite* sets of templates, expressed schematically with the help of variables; in the verification context, this is particularly relevant if arrays or heap are encoded with the help of uninterpreted functions.

Example 5 Suppose that the binary function H represents heap contents, with heap accesses $obj.field$ translated to $H(obj, field)$, and is used to state an interpolation problem:

$$(H(a, f) \doteq c \wedge H(b, g) \not\doteq null) \quad \wedge \quad (b \doteq c \wedge H(b, g) \doteq null \wedge H(H(a, f), g) \doteq null)$$

An obvious interpolant is the formula $I_1 = (H(b, g) \not\doteq null)$. We might want to avoid interpolants with direct heap accesses $H(\cdot, g)$, and instead prefer the pattern $H(H(\cdot, f), g)$. To find alternative interpolants, we can use the templates $\{H(H(x, f), g), a, b, c\}$, the first of which contains a schematic variable x. The resulting abstraction excludes I_1, but yields the interpolant $I_2 = (b \doteq c \rightarrow H(H(a, f), g) \not\doteq null)$.

Definition 7 (Schematic term abstraction) Suppose an interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, and a finite set $T = \{t_1[\bar{s}, \bar{x}_1], \ldots, t_n[\bar{s}, \bar{x}_n]\}$ of terms with free variables $\bar{x}_1, \ldots, \bar{x}_n$. The relation abstraction $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ defined by

$$R_A^T[\bar{s}', \bar{s}] = \bigwedge_{i=1}^{n} \forall \bar{x}_i.\ t_i[\bar{s}', \bar{x}_i] \doteq t_i[\bar{s}, \bar{x}_i], \qquad R_B^T[\bar{s}, \bar{s}''] = \bigwedge_{i=1}^{n} \forall \bar{x}_i.\ t_i[\bar{s}, \bar{x}_i] \doteq t_i[\bar{s}'', \bar{x}_i]$$

is called *schematic term interpolation abstraction* over T.

Note that schematic term interpolation abstractions reduce to ordinary term interpolation abstractions (as in Def. 4) if none of the template terms contains free variables.

Quantified abstractions are clearly less interesting for logics that admit quantifier elimination, such as PA, but they are relevant whenever uninterpreted functions (EUF) are involved.

Lemma 8 (Solvability in EUF) *Suppose* $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ *is an interpolation problem in EUF,* $T = \{t_1[\bar{s}, \bar{x}_1], \ldots, t_n[\bar{s}, \bar{x}_n]\}$ *a finite set of schematic terms, and* $f = \langle f_1, \ldots, f_n \rangle$ *a vector of fresh functions with arities* $|\bar{x}_1|, \ldots, |\bar{x}_n|$*, respectively. The abstract interpolation problem for* $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ *is solvable if and only if there is a formula* $I[f_1, \ldots, f_n]$ *(without non-logical symbols other than* $\bar{f}$*) such that* $I[t_1[\bar{s}, \cdot], \ldots, t_n[\bar{s}, \cdot]]$ *is an interpolant of* $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$.

The expression $I[t_1[\bar{s}, \cdot], \ldots, t_n[\bar{s}, \cdot]]$ denotes the formula obtained by replacing each occurrence of a function f_i in $I[f_1, \ldots, f_n]$ with the template $t_i[\bar{s}, \bar{x}_i]$, substituting the arguments of f_i for the schematic variables $\bar{x}_i$.

Proof "⇐": $I[t_1[\bar{s}, \cdot], \ldots, t_n[\bar{s}, \cdot]]$ is also an abstract interpolant, which implies that the abstract interpolation problem is solvable.

"⇒": observe that if the abstract interpolation problem is solvable, conjunction (5) is unsatisfiable:

$$\Big(A[\bar{s}_A, \bar{s}'] \wedge \psi_A\Big) \wedge \Big(\psi_B \wedge B[\bar{s}'', \bar{s}_B]\Big) \tag{5}$$

where

$$\psi_A = \bigwedge_{i=1}^{n} \forall \bar{x}_i.\ t_i[\bar{s}', \bar{x}_i] \doteq f_i(\bar{x}_i) \qquad \psi_B = \bigwedge_{i=1}^{n} \forall \bar{x}_i.\ f_i(\bar{x}_i) \doteq t_i[\bar{s}'', \bar{x}_i]$$

An interpolant $I[f_1, \ldots, f_n]$ can be computed from (5) using FOL interpolation techniques. □

6 The Algebra of Interpolation Abstractions

It is frequently useful to construct new interpolation abstractions from existing ones, for instance to combine term, inequality, and predicate interpolation abstractions. Combination is possible through several algebraic operations. For instance, given two interpolation abstractions (T_A, T_B) and (T'_A, T'_B), the composition $(T_A, T_B) \circ_L (T'_A, T'_B)$ constructs an abstract interpolation that composes the functions in the abstractions component-wise. New interpolation abstractions can be constructed similarly through conjunction, disjunction, and other operators. In the whole section we fix some logic, as well as a list $\bar{s}$ of common symbols.

6.1 Algebraic Properties of Interpolation Abstractions in General

In order to define algebraic operations, it is first necessary to introduce notions of implication and equivalence between interpolation abstractions:

Definition 8 (Ordering of Interpolation Abstractions) Let (T_A, T_B) and (T'_A, T'_B) be two interpolation abstractions. We say that (T_A, T_B) is *at least as strong* as (T'_A, T'_B), written $(T_A, T_B) \Rightarrow (T'_A, T'_B)$, if

(i) for any two formulae $A_1[\bar{s}_{A_1}, \bar{s}], A_2[\bar{s}_{A_2}, \bar{s}]$ with $\exists \bar{s}_{A_1}.\, A_1[\bar{s}_{A_1}, \bar{s}] \equiv \exists \bar{s}_{A_2}.\, A_2[\bar{s}_{A_2}, \bar{s}]$ and $A'_1[\bar{s}_{A'_1}, \bar{s}] = T_A(A_1[\bar{s}_{A_1}, \bar{s}])$ and $A'_2[\bar{s}_{A'_2}, \bar{s}] = T'_A(A_2[\bar{s}_{A_2}, \bar{s}])$, it is the case that the implication $\exists \bar{s}_{A'_1}.\, A'_1[\bar{s}_{A'_1}, \bar{s}] \Rightarrow \exists \bar{s}_{A'_2}.\, A'_2[\bar{s}_{A'_2}, \bar{s}]$ holds;

(ii) for any two formulae $B_1[\bar{s}, \bar{s}_{B_1}], B_2[\bar{s}, \bar{s}_{B_2}]$ with $\exists \bar{s}_{B_1}.\, B_1[\bar{s}, \bar{s}_{B_1}] \equiv \exists \bar{s}_{B_2}.\, B_2[\bar{s}, \bar{s}_{B_2}]$ and $B'_1[\bar{s}, \bar{s}_{B'_1}] = T_A(B_1[\bar{s}, \bar{s}_{B_1}])$ and $B'_2[\bar{s}, \bar{s}_{B'_2}] = T'_A(B_2[\bar{s}, \bar{s}_{B_2}])$, it is the case that the implication $\exists \bar{s}_{B'_1}.\, B'_1[\bar{s}, \bar{s}_{B'_1}] \Rightarrow \exists \bar{s}_{B'_2}.\, B'_2[\bar{s}, \bar{s}_{B'_2}]$ holds.

We say that (T_A, T_B) and (T'_A, T'_B) are *equivalent,* written $(T_A, T_B) \equiv (T'_A, T'_B)$, if it is the case that $(T_A, T_B) \Rightarrow (T'_A, T'_B)$ and $(T'_A, T'_B) \Rightarrow (T_A, T_B)$.

Note that $\equiv$ is *not* immediately an equivalence relation on interpolation abstractions, since an interpolation abstraction is not necessarily equivalent to itself ($\equiv$ is not reflexive): for instance, an abstraction might map equivalent, but syntactically distinct formulae to non-equivalent formulae. We therefore focus on the set L of all *self-equivalent* (or *extensive*) interpolation abstractions, for the fixed logic and symbols $\bar{s}$. In particular, relation abstractions

(Def. 3) are all self-equivalent. Since $\equiv$ is an equivalence relation on L, we can in the next paragraphs consider the set $L/\equiv$ of equivalence classes.

We can first observe that the set L is closed under the operations conjunction $\wedge_L$, disjunction $\vee_L$, identity Id_L, and top $\top_L$, defined in the following equations. Some care is required for the handling of local symbols: in the definition of $\wedge_L$ and $\vee_L$, fresh symbols introduced by the functions T_A, T_B, T'_A, T'_B (e.g., the $\bar{s}_{A'}$ in $A'[\bar{s}_{A'}, \bar{s}] = T_A(A[\bar{s}_A, \bar{s}])$) might clash and lead to incorrect results. Such clashes can be handled by appropriate renaming, which we assume is done implicitly.

$$\begin{aligned}
Id_L &:= (\lambda A.\, A,\ \lambda B.\, B) \\
\top_L &:= (\lambda A.\, \mathit{true},\ \lambda B.\, \mathit{true}) \\
(T_A, T_B) \wedge_L (T'_A, T'_B) &:= (\lambda A.\, T_A(A) \wedge T'_A(A),\ \lambda B.\, T_B(B) \wedge T'_B(B)) \\
(T_A, T_B) \vee_L (T'_A, T'_B) &:= (\lambda A.\, T_A(A) \vee T'_A(A),\ \lambda B.\, T_B(B) \vee T'_B(B))
\end{aligned}$$

All operations can be extended to the equivalence classes in $L/\equiv$, since $\equiv$ is a congruence relation. The resulting algebra forms a *bounded distributive lattice* [16] where Id_L is the bottom element, $\top_L$ is the top element and all elements are ordered by implication as in Def. 8.

In addition, L is closed under composition $\circ_L$, and has the structure of a monoid with binary operation $\circ_L$ and Id_L as the neutral element:

$$(T_A, T_B) \circ_L (T'_A, T'_B) := (\lambda A.\, T_A(T'_A(A)),\ \lambda B.\, T_B(T'_B(B)))$$

6.2 Algebraic Properties of Relation Abstractions

Similarly to the case of general interpolation abstractions, the subspace of *relation abstractions* (as in Def. 3) has the structure of both a lattice and a monoid. Since relation abstractions can be complemented, the lattice of relation abstractions is even a *Boolean lattice.*

We denote the set of relation abstractions $(R_A[\bar{s}', \bar{s}], R_B[\bar{s}, \bar{s}''])$ (over fixed and disjoint lists $\bar{s}, \bar{s}', \bar{s}''$ of symbols) by L_R. This set is again closed under a variety of operations:

$$\begin{aligned}
Id_R &:= (Id[\bar{s}', \bar{s}],\ Id[\bar{s}, \bar{s}'']) \\
\top_R &:= (\mathit{true},\ \mathit{true}) \\
(R_A, R_B) \wedge_R (R'_A, R'_B) &:= (R_A \wedge R'_A,\ R_B \wedge R'_B) \\
(R_A, R_B) \vee_R (R'_A, R'_B) &:= (R_A \vee R'_A,\ R_B \vee R'_B) \\
\neg_R(R_A, R_B) &:= (\neg R_A \vee Id[\bar{s}', \bar{s}],\ \neg R_B \vee Id[\bar{s}, \bar{s}''])
\end{aligned}$$

It can be observed that $\vee_R$ and $\vee_L$ coincide on relation abstractions; i.e., applying $\vee_L$ to relation abstractions yields a result equivalent to $\vee_R$. The same holds for Id_R and Id_L, as well as $\top_R$ and $\top_L$. In contrast, $\wedge_R$ behaves differently from $\wedge_L$; in general, the result of applying $\wedge_L$ to two relation abstractions produces an interpolation abstraction that cannot even be represented as a relation abstraction.

Relation abstractions can be ordered by implication: we write $(R_A, R_B) \Rightarrow (R'_A, R'_B)$ if $R_A \Rightarrow R'_A$ and $R_B \Rightarrow R'_B$, and $(R_A, R_B) \equiv (R'_A, R'_B)$ if $(R_A, R_B) \Rightarrow (R'_A, R'_B)$ and $(R'_A, R'_B) \Rightarrow (R_A, R_B)$. It can further be shown that $\equiv$ is a congruence relation for the above operations $Id_R, \top_R, \wedge_R, \vee_R, \neg_R$, and that the quotient $L_R/\equiv$ has the structure of a Boolean lattice.

7 Exploration of Interpolants

In a typical application scenario of our interpolation abstraction framework (e.g., in a model checker), we will not consider just a single fixed interpolation abstraction, but rather a whole *family* of such abstractions. Working with multiple interpolation abstractions turns out to be meaningful for several reasons: (i) for each interpolation problem we might want to compute multiple different interpolants, which can be achieved by successively applying several interpolation abstractions; (ii) by ranking interpolation abstractions, the quality of resulting interpolants can be controlled. For instance, in the example in Sect. 1.1, we consider interpolant I_2 constructed using templates $\{x_1 - i_1, j\}$ as "better" than interpolant I_1 for the template i_1; (iii) every individual interpolation abstraction is feasible for some interpolation problems, and infeasible for others. This necessitates the definition of a whole family of abstractions, so that some feasible abstractions can be picked for every interpolation problem.

To formalise this concept of *interpolant exploration,* we arrange families of interpolation abstractions as *abstraction lattices,* and present search algorithms on such lattices. As described in Sect. 6, interpolation abstractions have algebraic properties that can be used when defining such abstraction lattices. Abstraction lattices are equipped with a monotonic mapping μ to interpolation abstractions (T_A, T_B), ordered as in Def. 8. The following paragraphs focus on the case of *finite* abstraction lattices; the handling of infinite (parametric) abstraction lattices is planned as future work.

Definition 9 (Abstraction lattice) Suppose $\bar{s}$ is a list of non-logical symbols, for some arbitrary but fixed logic. An *abstraction lattice* is a pair $(\langle L, \sqsubseteq_L \rangle, \mu)$ consisting of a complete lattice $\langle L, \sqsubseteq_L \rangle$ and a monotonic mapping μ from elements of $\langle L, \sqsubseteq_L \rangle$ to interpolation abstractions (T_A, T_B) over $\bar{s}$, with the property that $\mu(\bot) = (Id_A, Id_B)$ is the identity abstraction (i.e., $Id_A(A) = A$ and $Id_B(B) = B$ for all formulae A, B).

Given an interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, the elements of an abstraction lattice that map to *feasible* interpolation abstractions form a downward closed set; an illustration is given in Fig. 3, where feasible elements are shaded in gray. Provided that the concrete interpolation problem is solvable, the set of feasible elements in the lattice is non-empty, due to the requirement that $\mu(\bot) = (Id_A, Id_B)$.

Particularly interesting are *maximal feasible* interpolation abstractions, i.e., the maximal elements within the set of feasible interpolation abstractions. Maximal feasible abstractions restrict interpolants in the strongest possible way, and are therefore most suitable for exploring interpolants; we refer to the set of maximal feasible elements within an abstraction lattice as *abstraction frontier*.

7.1 Construction of Abstraction Lattices

When working with interpolation abstractions generated by templates, abstraction lattices can naturally by constructed as the *powerset lattice* of some template base set (ordered by the superset relation); this construction applies to term, inequality, and predicate templates. Further, the operations introduced in Sect. 6 can be used to combine simple lattices into more sophisticated ones; for instance, a useful construction is to form the *product* of two lattices, defining the mapping μ as the pairwise conjunction, disjunction, or composition of the individual mappings μ_1, μ_2.

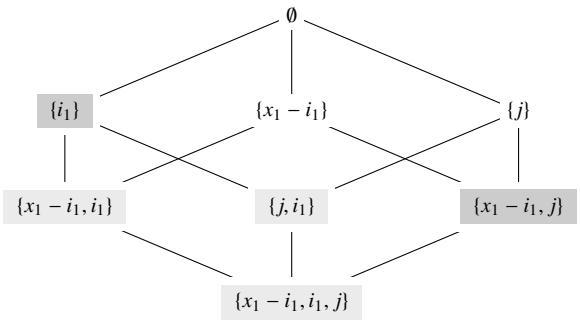


Fig. 3: The abstraction lattice for the running example. The light gray shaded elements are feasible, the dark gray ones maximal feasible.

Algorithm 1: Exploration algorithm

Input: Interpolation problem $P = A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, abstraction lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$
Result: Set of maximal feasible interpolation abstractions

```
Frontier ← ∅;
while ∃ feasible abs ∈ L, incomparable with all x ∈ Frontier do
    Frontier ← Frontier ∪ {maximise(P, abs)};
end
return Frontier;
```

Example 6 An abstraction lattice for the example in Sect. 1.1 is $(\langle \wp(T), \supseteq \rangle, \mu)$, with base templates $T = \{x_1 - i_1, i_1, j\}$ and μ mapping each element to the abstraction in Def. 4. Note that the bottom element of the lattice represents the full set T of templates (the weakest abstraction), and the top element the empty set $\emptyset$ (the strongest abstraction). Also, note that $\mu(T)$ is the identity abstraction (Id_A, Id_B), since T is a basis of the vector space of linear functions in x_1, i_1, j.

The lattice is presented in Fig. 3, with feasible elements in light gray. The maximal feasible elements $\{i_1\}$ and $\{x_1 - i_1, j\}$ map to interpolation abstractions with the abstract interpolants I_1 and I_2, respectively, as illustrated in Fig 2. Smaller feasible elements (closer to $\bot$) correspond to larger sub-lattices of abstract interpolants, and therefore provide weaker guidance for a theorem prover; for instance, element $\{j, i_1\}$ can produce all abstract interpolants that $\{i_1\}$ generates, but can in addition lead to interpolants like $I_3 = (j \neq 0 \vee i_1 \leq 49)$.

7.2 Computation of Abstraction Frontiers

In the case of abstraction lattices that are Boolean lattices, like the one in Fig. 3, the computation of abstraction frontiers can be carried out using algorithms for the well-known problem of computing *minimal unsatisfiable subsets* (e.g., [41]). Such algorithms do not immediately carry over, however, to non-Boolean lattices, which can also be relevant abstraction lattices. We therefore present a binary search-based algorithm to compute abstraction frontiers of arbitrary finite abstraction lattices. In later sections, this algorithm will be extended to also take *costs* into account, as a means to rank interpolation abstractions.

Algorithm 2: Maximisation algorithm *maximise*(P, abs)

Input: Interpolation problem $P = A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, feasible abstraction $abs \in L$
Result: Maximal feasible abstraction

```
1 while ∃ feasible immediate successor fs of abs do
2 |   pick element middle such that fs ⊑_L middle ⊑_L ⊤;
3 |   if middle is feasible then
4 |   |   abs ← middle;
5 |   else
6 |   |   abs ← fs;
7 |   end
8 end
9 return abs;
```

The search is described in Algorithms 1 and 2. Algorithm 1 describes the top-level procedure for finding maximal elements in an abstraction lattice. The algorithm repeatedly checks whether feasible abstractions $abs \in L$ exist that are incomparable with the maximum feasible abstractions found so far, i.e., such that no $x \in Frontier$ with $abs \sqsubseteq_L x$ or $x \sqsubseteq_L abs$ exists (line 2). Suitable methods for computing such incomparable elements can be defined based on the shape of the chosen abstraction lattice; for instance, if the abstraction lattice is a Boolean lattice, finding incomparable abstractions amounts to solving the problem of finding minimal *hitting sets* for the *Frontier* [24] (a hitting set is a set that has elements in common with every set in the *Frontier*). As long as incomparable elements can be found, they are maximised by calling the *maximise* function (described in Algorithm 2), and added to the frontier.

In Algorithm 2 we describe the procedure for finding a maximal feasible abstraction *mfa* with the property that $abs \sqsubseteq_L mfa$. In each iteration of the maximisation loop, it is checked whether *abs* has any feasible parents (line 1); if this is not the case, *abs* has to be maximal feasible and is returned. Otherwise, in the loop body the algorithm executes a binary search on the set of elements in between *abs* and $\top$. The algorithm depends on the ability to efficiently compute (random) middle elements between two elements $a \sqsubset b$ of the lattice (line 2); again, this functionality can best be implemented specifically for an individual lattice, and is not shown here.

It should be noted that checking the feasibility of an interpolation abstraction (T_A, T_B), for an interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, can be done by a simple check whether the conjunction $T_A(A[\bar{s}_A, \bar{s}]) \wedge T_B(B[\bar{s}, \bar{s}_B])$ is unsatisfiable (assuming a logic with the interpolation property). Repeating this check for a large number of abstractions can be optimised with the help of *incremental SMT:* typically, only a small part of the formula $T_A(A[\bar{s}_A, \bar{s}]) \wedge T_B(B[\bar{s}, \bar{s}_B])$ will actually depend on the abstraction (T_A, T_B), in particular for relation abstractions. Common conjuncts can therefore be factored out and handed over to an SMT solver upfront.

Lemma 9 (Correctness of exploration algorithm) *When applied to a finite abstraction lattice, Algorithm 1 terminates and returns the set of maximal feasible elements.*

Proof To see that the returned *Frontier* only contains maximal feasible abstractions, note that algorithm *maximise*(P, abs) only returns abstractions that are feasible, and only abstractions without feasible successors (i.e., maximal feasible ones). The returned *Frontier* contains *all* maximal feasible abstractions, since any missing maximal feasible abstractions $mfa \notin Frontier$ would have to be incomparable with the elements in *Frontier* (due to maximality), and thus the loop condition in Algorithm 1, line 2 holds.

Algorithm 1 terminates, since the considered abstraction lattice is finite, and the set *Frontier* grows by one element in every iteration of the while loop. Namely, assume that in some iteration an abstraction $\mathit{maximise}(P, \mathit{abs})$ is produced that is already an element of *Frontier*; in this case, $\mathit{abs} \sqsubseteq_L \mathit{maximise}(P, \mathit{abs})$ cannot have been incomparable with *Frontier*. Algorithm 2 terminates since finite lattices have finite height, and *abs* grows strictly in every iteration of the while loop. □

A useful refinement of the exploration algorithm is to *canonise* lattice elements during search. Elements $a, b \in L$ are considered equivalent if they are mapped to (logically) equivalent abstraction relations by μ. Canonisation can select a representative for every equivalence class of lattice elements, and search be carried out only on such canonical elements.

7.3 Guiding Interpolant Exploration with Costs

Given an abstraction frontier, it is possible to compute a range of interpolants solving the original interpolation problem. However, for large abstraction frontiers this may be neither feasible nor necessary. It is more useful to define a measure for the quality of interpolation abstractions, again exploiting domain-specific knowledge, and only use the best abstractions for interpolation.

To select good maximal feasible interpolation abstractions, we define an *anti-monotonic* cost function $\mathit{cost} : L \to \mathbb{N}$ that maps elements of an abstraction lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$ to a natural number, with lower values indicating that an interpolation abstraction is considered better. The anti-monotonicity property $(\forall a, b \in L.\, a \sqsubseteq_L b \Rightarrow \mathit{cost}(a) \geq \mathit{cost}(b))$ encompasses that coarser abstractions (higher up in the lattice) have lower cost. In the case of abstractions constructed using a powerset lattice over templates ($L = \wp(T)$), it is natural to assign a cost to every element in T ($\mathit{cost} : T \to \mathbb{N}$), and to define the cost of a lattice element $A \in L$ as $\mathit{cost}(A) = \sum_{t \in A} \mathit{cost}(t)$. Similarly, for product lattices the cost function can be computed as the sum of the costs of the components.

Our abstraction lattice in Fig. 3 has two maximal feasible abstractions, $\{i_1\}$ and $\{x_1 - i_1, j\}$, which result in computing the interpolants I_1 and I_2, respectively. We can define a cost function that assigns a high cost to $\{i_1\}$ and a low cost to $\{x_1 - i_1, j\}$, expressing the fact that we prefer to not talk about the loop counter i_1 in absolute terms. More generally, assigning a high cost to variables representing loop counters is a reasonable strategy for obtaining general interpolants (a similar observation is made in [2], and implemented with the help of "term abstraction").

Once a cost function has been defined, the goal is to compute those abstractions from the *Frontier* set that have minimal cost. Naively, this can be done by first computing the whole *Frontier* set, using Algorithms 1 and 2, and then removing those elements that are too costly; however, for realistic abstraction lattices this procedure tends to be slow. Instead, it is possible to exploit costs already during search, eagerly pruning away those parts of the search space that cannot contain abstractions with low cost. We describe an optimisation to the exploration algorithms that uses costs to this effect in Algorithms 3 and 4.

Besides *Frontier*, in Algorithm 3 an additional set of *costly* abstractions (*CostlyAbs*) is maintained. A costly abstraction c is one whose cost $\mathit{cost}(c)$ has been identified as being greater than the minimal cost of feasible abstractions, and that has the property that none of its successors is feasible; as a consequence, the part of the abstraction lattice above c cannot contain low-cost frontier elements.

Algorithm 3: Optimised Exploration Algorithm

Input: Interpolation problem $P = A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, abstraction lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$
Result: Set of *all* maximal feasible interpolation abstractions of minimal cost

```
1  CostlyAbs ← ∅;
2  Frontier ← ∅;
3  minCost ← ∞;
4  while ∃ feasible abs ∈ L, incomparable with Frontier and CostlyAbs do
5      m or c ← boundedMaximise(P, abs, minCost);
6      if m was returned, and cost(m) < minCost then
7          CostlyAbs ← CostlyAbs ∪ Frontier;
8          Frontier ← {m};
9          minCost ← cost(m);
10     else
11         Frontier ← Frontier ∪ {m} or CostlyAbs ← CostlyAbs ∪ {c};
12     end
13 end
14 return Frontier;
```

Algorithm 4: Optimised maximisation algorithm $boundedMaximise(P, abs, minCost)$

Input: Interpolation problem $P = A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, feasible abstraction $abs \in L$, minimal cost bound *minCost*
Result: $m \in L$ s.t. $abs \sqsubseteq_L m$, m is maximal feasible, and $cost(m) \leq minCost$ **or**
$c \in L$ s.t. $abs \sqsubseteq_L c$, $cost(c) > minCost$, and all successors of c are infeasible

```
1  upperBound ← ⊤;
2  while true do
3      fs ← undef;
4      for all immediate successors s of abs, while fs is undefined do
5          if s ⊑_L upperBound then
6              if s is feasible then
7                  fs ← s;
8              else if ∃b. feasibilityBound(abs, s, b) then
9                  upperBound ← upperBound ⊓ b;
10                 if cost(upperBound) > minCost then
11                     return c ← upperBound;
12                 end
13                 if upperBound is feasible then
14                     return m ← upperBound;
15                 end
16             end
17         end
18     end
19     if fs is defined then
20         pick abstraction middle such that fs ⊑_L middle ⊑_L upperBound;
21         if middle is feasible then
22             abs ← middle;
23         else
24             abs ← fs;
25         end
26     else
27         if cost(abs) > minCost then
28             return c ← abs;
29         else
30             return m ← abs;
31         end
32     end
33 end
```

The generalised maximisation function (*boundedMaximise*, Algorithm 4) returns either a maximal feasible abstraction m of minimal cost, or it returns a costly abstraction c (which may or may not be feasible). Feasible abstractions of minimal cost are added to the *Frontier*, while costly abstractions are added to the *CostlyAbs* set. If a returned maximal feasible abstraction improves upon the current cost bound (defined by the *minCost* variable), then the *minCost* variable is updated with the new minimal cost, and all previous frontier abstractions are moved to *CostlyAbs*.

Like Algorithm 2, Algorithm 4 proceeds by increasing the abstraction *abs* until an abstraction is reached whose successors are all infeasible. To this end, the `for` loop (line 4) iterates over the immediate successors of *abs*; if a feasible successor is found, the loop is left, while knowledge about infeasible successors is used to improve the *upperBound* variable.

The algorithm maintains the invariant that *abs*, and all of its feasible successors are below *upperBound*. If it is detected that $cost(upperBound) > minCost$, it follows (thanks to anti-monotonicity of *cost*) that no feasible abstractions with low cost can exist above *abs*, and the algorithm can return immediately. In this way, the search space can be pruned significantly.

In order to update the variable *upperBound* (line 8), the algorithm exploits the fact that a feasible abstraction *abs* with an infeasible successor s has been found. Given the pair *abs*, s, we call an element $b \in L$ a *feasibility bound* if the following properties are satisfied:

$$feasibilityBound(abs, s, b) \equiv \begin{cases} abs \text{ is feasible and } s \text{ is infeasible,} \\ abs = s \sqcap b, \text{ and} \\ \text{for every feasible abstraction } x \text{ with } abs \sqsubseteq x \\ \qquad \text{it holds that } x \sqsubseteq b\ . \end{cases}$$

In other words, given a feasible abstraction *abs* with infeasible successor s of *abs*, the predicate *feasibilityBound* provides an upper bound b for every feasible successor of *abs*. This implies that subsequent maximisation can ignore parts of the lattice that are not underneath b.

The existence of upper bounds b is determined by the considered lattice. In the special case that the abstraction lattice is a *distributive lattice* (e.g., a powerset lattice), a simpler definition of feasibility bounds can be used:

$$feasibilityBound_{dist}(abs, s, b) \equiv \begin{cases} abs \text{ is feasible and } s \text{ is infeasible,} \\ abs = s \sqcap b, \text{ and} \\ b \text{ is a direct predecessor of } \top\ . \end{cases}$$

Since it can be observed that $feasibilityBound_{dist}(abs, s, b)$ implies the previous predicate $feasibilityBound(abs, s, b)$, for distributive lattices, the former can be used as a more effective and sufficient condition.

Lemma 10 (Correctness of optimised exploration algorithm) *When applied to a finite abstraction lattice, Algorithm 3 terminates and returns the set of minimal cost, maximal feasible abstractions.*

Proof Note that the outer loops of the algorithms have the following loop invariants:

$$\begin{aligned} Inv_{alg3} = {} & \forall x \in Frontier.\ (cost(x) = minCost \wedge x \text{ is maximal feasible }) \\ & \wedge\ \forall x \in CostlyAbs.\ (cost(x) > minCost \wedge \text{ all successors of } x \text{ are infeasible }) \\ Inv_{alg4} = {} & \forall x \in L.\ (abs \sqsubseteq_L x \wedge x \text{ is feasible } \Rightarrow x \sqsubseteq_L upperBound) \\ & \wedge\ abs \text{ is feasible} \end{aligned}$$

It follows directly that *Frontier* in Algorithm 3 can only contain maximal feasible abstractions. Further, upon termination the *Frontier* contains *all* maximal feasible abstractions with minimal cost. Namely, assume that there is a maximal feasible abstraction $mfa \notin Frontier$ with $cost(mfa) \leq minCost$. As in the proof of Lem. 9, it follows that *mfa* is incomparable with *Frontier*. Further, *mfa* cannot be above any element in *CostlyAbs*, since successors of *CostlyAbs* are infeasible; *mfa* cannot be below any element in *CostlyAbs* due to anti-monotonicity of *cost*. Therefore the loop condition must be satisfied, contradicting the assumption that Algorithm 3 had terminated.

Termination of Algorithm 3 can be shown like in the proof of Lem. 9.

Partial correctness of Algorithm 4 follows from its loop invariant. Termination is guaranteed since finite lattices have finite height, and *abs* grows strictly in every iteration of the `while` loop while *upperBound* may only decrease strictly with every iteration. Further, since the invariant holds that $abs \sqsubseteq_L upperBound$ Algorithm 4 terminates. □

8 Experimental Evaluation in Model Checking

Interpolation abstraction can be applied whenever Craig interpolation is used by a model checker to eliminate spurious counterexamples. We evaluate the effectiveness of the technique in two ways: using existing benchmarks in the form of recursive Horn clauses (Sect. 8), representing various forms of verification tasks, and by integration into a new model checker for the analysis of Petri net models (Sect. 9).

8.1 Sequence Interpolation Abstraction

Predicate abstraction-based model checkers usually consider more general interpolation problems than just binary Craig interpolation: the interpolation queries might concern sequences of interpolants, tree interpolants, or other interpolation schemata [49]. For instance, given an unsatisfiable conjunction $A_1 \wedge \ldots \wedge A_n$ (in practice, often corresponding to an infeasible path in a program), an *interpolant sequence* [29,43] is a sequence $I_0, I_1, \ldots, I_n$ of formulae such that

(i) $I_0 = true$, $I_n = false$,
(ii) for all $i \in \{1, \ldots, n\}$, the implication $I_{i-1} \wedge A_i \Rightarrow I_i$ holds, and
(iii) for all $i \in \{0, \ldots, n\}$, the non-logical symbols in I_i occur in both $A_1 \wedge \cdots \wedge A_i$ and $A_{i+1} \wedge \cdots \wedge A_n$.

The framework of interpolation abstraction can be extended to such generalised forms of interpolation. We only explain the case of interpolant sequences at this point, but other interpolation schemata can be handled in a similar way.

Assume that $\bar{s}_1, \ldots, \bar{s}_n$ are the shared non-logical symbols in $A_1, \ldots, A_n$, respectively; i.e., each $\bar{s}_i$ is a list of the common symbols of A_i and $A_1 \wedge \cdots \wedge A_{i-1} \wedge A_{i+1} \wedge \cdots \wedge A_n$.

Definition 10 (n-ary interpolation abstraction) Suppose $\bar{s}_1, \ldots, \bar{s}_n$ are lists of non-logical symbols. An *n-ary interpolation abstraction* is a tuple $\langle T_1, \ldots T_n \rangle$ of functions mapping formulae to formulae, with the following properties:

(i) for $i \in \{1, \ldots, n\}$ and any formula $A_i[\bar{s}_i, \bar{t}_i]$, and $\bar{t}_i$ disjoint from $\bar{s}_i$, the result of applying T_i is a new formula $A'_i[\bar{s}_i, \bar{t}'_i] = T_i(A_i[\bar{s}_i, \bar{t}_i])$ such that $(\exists \bar{t}_i.\, A_i[\bar{s}_i, \bar{t}_i]) \Rightarrow (\exists \bar{t}'_i.\, A'_i[\bar{s}_i, \bar{t}'_i])$.

(ii) for $i, j \in \{1, \ldots, n\}$ with $i \neq j$, and formulae $A_i[\bar{s}_i, \bar{t}_i]$ and $A_j[\bar{s}_j, \bar{t}_j]$ such that $\bar{t}_i, \bar{t}_j$, and $\bar{s}_i \cup \bar{s}_j$ are pairwise disjoint, the common non-logical symbols of $T_i(A_i[\bar{s}_i, \bar{t}_i])$ and $T_j(A_j[\bar{s}_j, \bar{t}_j])$ are contained in $\bar{s}_i$ (and therefore also in $\bar{s}_j$).

We call $T_1(A_1) \wedge \cdots \wedge T_n(A_n)$ an *abstract sequence interpolation problem*.

In the same way as before (Lem. 1), it can be shown that every solution of an abstract sequence interpolation problem is also a solution of the concrete problem $A_1 \wedge \ldots \wedge A_n$, but in general not vice versa.

In practice, n-ary interpolation abstractions can be derived from binary interpolation abstractions, which can be applied at an arbitrary *abstraction point* in a sequence interpolation problem $A_1 \wedge \cdots \wedge A_i \wedge A_{i+1} \wedge \cdots \wedge A_n$. If $\bar{s}$ are the common non-logical symbols of A_i and A_{i+1}, and (T_A, T_B) is a binary interpolation abstraction for $\bar{s}$ (according to Def. 2), then $\langle Id, \ldots, Id, T_A, T_B, Id, \ldots, Id \rangle$ (with T_A at position i) is an n-ary interpolation abstraction, provided that T_A, T_B satisfy condition (ii) of Def. 10 (T_A and T_B must not introduce new non-logical symbols that clash with existing symbols in $A_1, \ldots, A_n$). This means that all interpolation abstractions introduced in Sect. 5 can also applied for sequence interpolation. It is also possible to combine multiple binary interpolation abstractions, applied at different abstractions points, into a single sequence interpolation abstraction.

The construction in the previous paragraph works for any sequence interpolation problem; however, in practice it is most meaningful for interpolation problems with the additional property that non-logical symbols are only shared between adjacent conjuncts: A_i and A_j only share symbols if $|i - j| \leq 1$. This additional requirement is meaningful because the application of an abstraction function T_A or T_B might otherwise not catch all occurrences of symbols shared between the conjuncts A_i, A_{i+1}. Sequence interpolation problems can be normalised to problems with shared symbols only between adjacent conjuncts by renaming symbols and adding further equations.

8.2 Experiments with Interpolation Abstraction for Solving Horn Clauses

We have integrated our technique into the predicate abstraction-based model checker Eldarica [33], which uses Horn clauses to represent different kinds of verification problems [26], and solves recursion-free Horn constraints (over linear integer arithmetic) to synthesise new predicates for abstraction [50]. As abstraction points, recurrent control locations representing loop heads in counterexamples are chosen, corresponding to recurrent relation symbols of Horn clauses. We then use Algorithm 1 to search for maximal feasible interpolation abstractions in the Cartesian product of the chosen abstraction lattices. With the help of cost functions, the best maximal feasible abstractions are determined, and subsequently used to compute abstract interpolants.

We compare four different abstractions lattices in our experiments; those lattices are pre-computed for each loop head h in a set of Horn constraints, and are always constructed as powerset lattices $(\langle \wp(T_h), \supseteq \rangle, \mu_h)$ over some set T_h of template terms or inequalities. The templates are chosen on the basis of simple static analyses to determine dataflow in the Horn constraints:

- **ABS (1)** uses finite term interpolation abstractions (Sect. 5.1), where the set T_h of templates is simply chosen to be the set of all program variables (corresponding to all arguments of a relation symbol in Horn constraints). A cost function $cost_h$ is used to distinguish different roles of a variable:

	Eldarica						Z3
	Norm	**DI**	**ABS (1)**	**ABS (2)**	**ABS (3)**	**ABS (4)**	
1. NT Driver benchmarks from CPAchecker [5]							
4 sat	2/**45.1**/**45.1**	2/56.8/56.8	2/71.9/71.9	2/ 108/ 108	2/ 125/ 125	2/ 113/ 113	**4**/ 545/ 707
6 unsat	3/ **162**/ **162**	3/ 418/ 301	3/ 589/ 751	2/ 200/ 200	2/ 264/ 264	3/ 378/ 394	**4**/ 374/ 303
2. SSH-Simplified benchmarks from CPAchecker [5]							
14 sat	**14**/**29.3**/24.8	**14**/ 106/ 101	**14**/94.9/88.5	**14**/ 119/ 123	13/ 151/ 133	**14**/ 144/ 142	10/ 101/**23.3**
9 unsat	9/12.6/11.4	9/35.2/34.0	9/32.4/14.8	9/38.5/24.8	9/39.7/25.1	9/50.7/21.8	9/ **5.7**/ **3.5**
3. Locks benchmarks from CPAchecker [6]							
11 sat	9/ 147/22.0	**11**/15.9/20.9	9/ 131/19.0	9/ 145/20.3	9/ 148/22.1	9/ 158/23.4	**11**/ **1.9**/ **1.5**
2 unsat	2/ 2.6/ 2.6	2/ 8.6/ 8.6	2/ 2.7/ 2.7	2/ 2.4/ 2.4	2/ 3.1/ 3.1	2/ 3.2/ 3.2	2/ **0.8**/ **0.8**
4. Benchmarks from "A Practical and Complete Approach to Predicate Refinement" [36]							
2 sat	1/ **0.1**/ **0.1**	1/ 0.1/ 0.1	1/ 0.1/ 0.1	**2**/ 0.4/ 0.4	**2**/ 0.3/ 0.3	**2**/ 1.2/ 1.2	0/ –/ –
10 unsat	10/19.5/ 0.3	10/20.1/ 0.3	10/ 125/ 1.8	10/92.5/ 1.7	10/ 132/ 1.8	10/ 132/ 6.4	10/ **0.4**/ **0.1**
5. Numerical Recipe benchmarks from "Synthesizing Software Verifiers from Proof Rules" [26]							
12 sat	9/12.6/ 1.7	9/15.6/ 1.9	11/15.7/ 2.2	**12**/64.7/ 4.1	11/18.0/ 4.9	11/21.9/ 4.9	11/ **9.7**/ **1.7**
1 unsat	0/ –/ –	**1**/49.8/49.8	**1**/22.2/22.2	**1**/**17.2**/**17.2**	**1**/29.8/29.8	**1**/29.7/29.7	0/ –/ –
6. Benchmarks from "Consistency analysis of decision-making programs" [13][4]							
29 sat	**29**/10.7/10.9	**29**/17.8/16.0	**29**/ **6.0**/ **5.0**	**29**/11.1/ 7.6	**29**/ 7.0/ 5.6	**29**/11.3/ 8.9	27/16.2/ 7.0
27 unsat	27/ 0.8/ 0.1	27/ 0.5/ 0.2	27/ 0.9/ 0.3	27/ 0.9/ 0.2	27/ 0.8/ 0.2	27/ 1.2/ 0.3	28/ **0.5**/ **0.1**
7. Benchmarks from "Dual Analysis for Proving Safety and Finding Bugs" [45]							
5 sat	**5**/76.9/ 3.6	**5**/36.9/ 8.9	**5**/84.8/ 3.8	4/12.7/ 8.1	**5**/82.4/ 6.0	**5**/ 124/ 7.1	4/ **0.2**/ **0.2**
1 unsat	1/ 6.5/ 6.5	1/ 7.2/ 7.2	1/ 8.4/ 8.4	1/ 6.4/ 6.4	1/ 8.3/ 8.3	1/ 8.3/ 8.3	1/ **0.5**/ **0.5**
8. Benchmarks from "Algorithmic Verification of Asynchronous Programs" [23]							
9 sat	9/20.8/ 0.4	9/20.2/ 0.5	9/ **8.4**/ 0.5	9/ 9.4/ 0.4	9/ 9.4/ 0.6	9/ 9.0/ 0.7	9/44.8/ **0.2**
11 unsat	11/ 0.8/ 0.4	11/ 1.2/ 0.4	11/ 0.8/ 0.5	11/ 0.8/ 0.4	11/ 0.9/ 0.4	11/ 0.9/ 0.5	11/ **0.3**/ **0.2**
9. Benchmarks from "Automatic Verification of Integer Array Programs" [9]							
6 sat	5/ 0.5/ 0.3	5/ 0.4/ 0.3	**6**/ 0.8/ 0.7	**6**/ 0.6/ 0.6	**6**/ 0.8/ 0.7	**6**/ 0.8/ 0.7	**6**/ **0.2**/ **0.2**
2 unsat	2/ 0.6/ 0.6	2/ 0.6/ 0.6	2/ 0.9/ 0.9	2/ 0.7/ 0.7	2/ 0.9/ 0.9	2/ 0.9/ 0.9	2/ **0.2**/ **0.2**
10. Benchmarks from "Programs with Lists Are Counter Automata" [8]							
2 sat	1/ 0.2/ 0.2	1/ **0.2**/ **0.2**	1/ 0.2/ 0.2	**2**/ 1.3/ 1.3	**2**/ 0.3/ 0.3	**2**/ 0.9/ 0.9	0/ –/ –
7 unsat	7/ 0.7/ 0.3	7/ 0.7/ 0.3	7/ 0.6/ 0.3	7/ 0.7/ 0.3	7/ 0.8/ 0.3	7/ 0.8/ 0.4	7/ **0.1**/ **0.1**
11. Benchmarks from "Inductive Invariant Generation via Abductive Inference" [17]							
46 sat	31/ 9.6/ 0.1	32/ 9.3/ 0.1	41/ 8.5/ 0.1	41/ 3.1/ 0.2	42/ **2.0**/ 0.2	**43**/36.7/ 0.2	37/27.9/ **0.0**
0 unsat	–/ –/ –	–/ –/ –	–/ –/ –	–/ –/ –	–/ –/ –	–/ –/ –	–/ –/ –
12. Regression benchmarks from "Automating Regression Verification" [20]							
54 sat	28/ 0.3/ 0.1	31/22.2/ **0.1**	41/ 0.3/ 0.1	45/24.1/ 0.1	**46**/ 4.0/ 0.2	43/ 1.3/ 0.5	6/ **0.1**/ 0.1
18 unsat	18/ 0.6/ 0.1	18/ 0.6/ 0.1	18/ 3.6/ 0.1	18/10.3/ 0.2	18/ 8.9/ 0.2	18/ 5.8/ 0.2	18/ **0.5**/ **0.0**
13. VHDL models from "Verifying Parametrised Hardware Designs Via Counter Automata" [54]							
4 sat	4/ 4.8/ 3.3	4/12.3/ 5.9	4/ 4.6/ 3.5	4/ 4.5/ 2.6	4/ **4.5**/ 2.9	4/ 5.7/ 5.0	4/ 6.8/ **0.2**
1 unsat	1/ 321/ 321	1/ 363/ 363	1/ 389/ 389	1/ 327/ 327	1/ 397/ 397	1/ 551/ 551	1/ **108**/ **108**
14. Timed automata benchmarks from "Horn Clauses for Communicating Timed Systems" [34]							
5 sat	5/10.4/ 1.8	5/33.3/ 1.7	5/ 9.8/ 2.9	5/ 8.5/ 4.2	5/12.8/ 3.4	5/11.0/ 3.4	5/ **8.0**/ **0.8**
26 unsat	26/ 0.6/ 0.3	26/ 0.6/ 0.3	26/ 1.1/ 0.6	26/ 0.7/ 0.3	26/ 1.3/ 0.6	26/ 1.3/ 0.7	26/ **0.3**/ **0.2**
15. Recursive Eldarica benchmarks							
18 sat	14/ 0.3/ 0.1	14/ 0.2/ 0.1	14/ 0.2/ 0.1	15/ 1.8/ 0.1	14/ 0.2/ 0.1	**16**/ 1.1/ 0.1	8/ **0.1**/ **0.0**
0 unsat	–/ –/ –	–/ –/ –	–/ –/ –	–/ –/ –	–/ –/ –	–/ –/ –	–/ –/ –
16. NECLA benchmarks							
3 sat	3/ 0.2/ 0.2	3/ 0.1/ 0.1	3/ 0.2/ 0.2	3/ 0.2/ 0.2	3/ 0.1/ 0.2	3/ 0.2/ 0.2	3/ **0.1**/ **0.1**
2 unsat	2/ 0.1/ 0.1	2/ 0.2/ 0.2	2/ 0.1/ 0.1	2/ 0.1/ 0.1	2/ 0.1/ 0.1	2/ 0.1/ 0.1	2/ **0.0**/ **0.0**
17. Benchmarks provided by D. Monniaux, personal communication							
5 sat	2/14.1/14.1	2/33.4/33.4	4/27.2/ 2.2	**5**/29.4/ 2.6	**5**/ **9.4**/ 2.6	4/14.4/ 6.0	**5**/79.6/ **0.2**
0 unsat	–/ –/ –	–/ –/ –	–/ –/ –	–/ –/ –	–/ –/ –	–/ –/ –	–/ –/ –
Total							
229 sat	171/**18.9**/ 0.4	177/23.0/ 0.5	199/20.3/ 0.5	**207**/28.4/ 0.8	**207**/23.4/ 0.6	**207**/32.5/ 1.7	150/37.7/ **0.4**
123 unsat	119/**10.0**/ 0.3	120/18.9/ 0.3	120/32.2/ 0.5	119/19.1/ 0.4	119/24.2/ 0.5	120/30.7/ 0.6	**121**/14.0/ **0.2**

Table 1: Experiments with Horn benchmarks over linear integer arithmetic (LIA). Each entry in the table specifies ⟨number of solved problems⟩/⟨average time (sec)⟩/⟨median time (sec)⟩. The columns correspond to standard Eldarica (**Norm**), Eldarica with disjunctive interpolation (**Norm**), Eldarica with the different interpolation abstractions (**ABS (1)–(4)**), and **Z3**. All experiments were done on an Intel Core i7 Duo 2.9 GHz with 8GB of RAM, with a timeout of 1000 seconds.

- $cost_h(x) = 1$ if x is a variable that is not assigned to in the loop with head h;
- $cost_h(x) = 9$ if x is incremented by exactly 1 in each loop iteration (a loop counter);
- $cost_h(x) = 4$ for all other variables x.

ABS (1) closely corresponds to term abstractions as in [2,57], and makes a model checker prefer interpolants with variables whose value does not change during execution of loops.

- **ABS (2)** also uses finite term interpolation abstractions, but computes a richer set of template terms. First a set $MVar_h$ of variables that are assigned to in a loop is computed. Assuming this set is non-empty, and $c \in MVar_h$ is some modified variable, the templates are chosen as

$$T_h \;=\; \{x \mid x \text{ a program variable}\} \;\cup\; \{x - c, x + c \mid x \in MVar_h \setminus \{c\}\}$$

with the cost function

$$cost_h(t) \;=\; \begin{cases} 1 & \text{if } t = x \text{ is a variable not modified in the loop,} \\ 6 & \text{if } t = x \text{ is a variable modified in the loop,} \\ 2 & \text{if } t = x - c, \\ 7 & \text{if } t = x + c. \end{cases}$$

The rationale is that templates $x-c, x+c$ can be combined to express arbitrary difference and octagonal properties between variables assigned in the loop (for instance, $x - y = (x - c) - (y - c)$); nevertheless, the set T_h only grows linearly in the overall number of program variables. The cost function is chosen such that templates expressing linear relationships between modified loop variables are preferred over individual variables.

- **ABS (3)** is an extension of ABS (2) that takes arithmetic progression of variables in the loop into account. We first compute a stride set

$$S \;=\; \{(x, \alpha) \mid x \text{ is a variable that is incremented by } \alpha \text{ on some path through the loop}\}$$

by means of static analysis; for instance, if x is consistently incremented by 1 on some path, and -2 on some other path, then $\{(x, 1), (x, -2)\} \subseteq S$. As a convention, if the loop also has paths on which a variable x progresses non-arithmetically (x is modified, but not incremented or decremented by a constant value), we assume $\{(x, 1), (x, -1)\} \subseteq S$. We then pick some fixed variable c that is assigned to in the loop, and choose templates

$$T_h \;=\; \{x \mid x \text{ a program variable}\} \;\cup\; \{\alpha_c x - \alpha_x c \mid (x, \alpha_x) \in S,\ (c, \alpha_c) \in S,\ x \neq c\}$$

and the cost function

$$cost_h(t) \;=\; \begin{cases} 1 & \text{if } t = x \text{ is a variable not modified in the loop,} \\ 6 & \text{if } t = x \text{ is a variable modified in the loop,} \\ 2 & \text{if } t = \alpha_c x - \alpha_x c. \end{cases}$$

- **ABS (4)** computes the same templates as ABS (3), but uses them to define finite inequalities interpolation abstractions (Sect. 5.2); i.e., for every template it is possible to separately consider upper and lower bounds.

All interpolation abstractions are implemented in the Eldarica Horn solver;[5] for our experiments, we used version v1.0-rc,[6] in which the abstractions can be enabled with the

[4] Z3 incorrectly classified one of the benchmarks from [13] as unsat instead of sat; this error was confirmed by the Z3 authors, and will be fixed in future versions of Z3.

[5] `https://github.com/uuverifiers/eldarica`

[6] `https://github.com/uuverifiers/eldarica/releases/download/v1.0-rc/eldarica-bin-2014-08-20.zip`

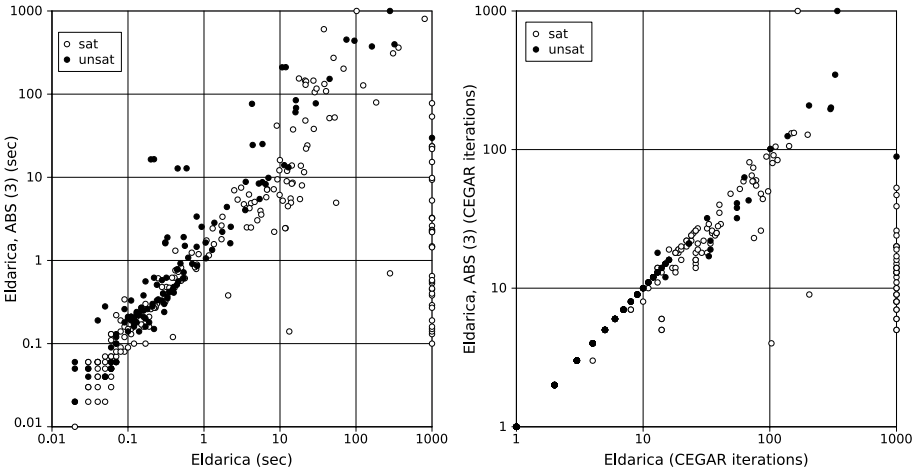


Fig. 4: Scatter plots comparing runtime and number of CEGAR iterations for normal Eldarica, and Eldarica with interpolation abstraction ABS (3).

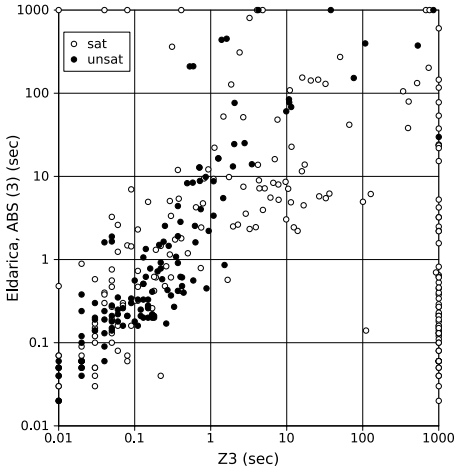


Fig. 5: Scatter plot comparing Z3 runtime with Eldarica ABS (3).

options `-abstract:term` for ABS (1), `-abstract:oct` for ABS (2), `-abstract:relEqs` for ABS (3), and `-abstract:relIneqs` for ABS (4).

The interpolation abstraction techniques were evaluated on altogether 352 publicly available benchmarks from the literature.[7] As a comparison, we also applied standard Eldarica to the benchmarks, as well as Eldarica with the disjunctive interpolation method from [50]. Finally, we compared to the Horn engine of Z3 [30] (using a beta version of Z3 4.3.2 checked out in January 2014), which is considered state-of-the-art and widely used for verification. The main results of our experiments are given in Table 1, and categorised according to the

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

benchmark source and the status (satisfiable/unsatisfiable). In addition, Fig. 4 shows scatter plots for standard Eldarica against ABS (3), with respect to runtime and the number of CEGAR iterations, and Fig. 5 gives a scatter plot comparing the runtime of ABS (3) with that of Z3.

It can first be observed that all methods can solve almost all unsatisfiable benchmarks. Interpolation abstraction is not beneficial for solving such benchmarks, but at the same time only causes minor slow-down compared to standard Eldarica (e.g., Fig. 4). Z3 exhibits the best performance on unsatisfiable benchmarks, but can altogether only solve one more benchmark than standard Eldarica.

The differences are more significant for satisfiable benchmarks, where standard Eldarica can solve more problems than Z3 (171 vs. 150), and Eldarica with interpolation abstraction (in particular ABS (2)–(4)) can solve many more problems than standard Eldarica (207 vs. 171). ABS (3) is the configuration that can solve the largest number of benchmarks in shortest average and median time, and also consistently needs fewer CEGAR iterations than standard Eldarica (Fig. 4). However, as the scatter plots in Fig. 4 and 5 show, the runtime of standard Eldarica, ABS (3), and Z3 are all comparable on satisfiable benchmarks. The richest family of interpolation abstractions, ABS (4), overall cannot solve more problems than ABS (3), but is significantly slower; this indicates that ABS (3) is indeed a good compromise between expressiveness and performance. However, ABS (4) performs well on some individual families of benchmarks (11 and 15).

Eldarica with disjunctive interpolation can solve more problems than standard Eldarica (177 vs. 171), but is not as effective as interpolation abstraction. However, disjunctive interpolation is on average significantly faster than other Eldarica configurations on satisfiable benchmarks in family 3.

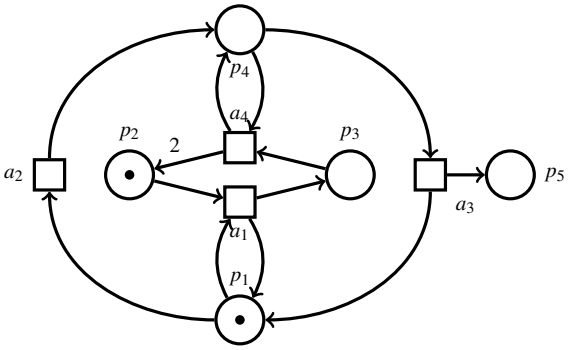
Overall, it can be concluded that interpolation abstractions only incur a small runtime overhead, and significantly extend the set of problems that can be solved using either interpolation methods or the generalised property-directed reachability algorithm of Z3.

9 A Case Study: Abstraction-based Analysis of Petri Nets

The previous sections evaluated the effectiveness of interpolation abstractions for a variety of verification problems. The considered interpolation abstractions were rather simple and, albeit inspired by criteria derived from software programs with numeric variables, general-purpose in the sense that they could be applied to arbitrary problems in the form of Horn constraints. We now give a case study on interpolation abstractions in the context of Petri nets, where knowledge about the semantics and structure of analysed systems can be exploited to construct domain-specific and more powerful interpolation abstractions.

9.1 Preliminaries

A Petri net is a model equipped with a finite set of counters, usually called places, ranging over the natural numbers. A configuration of a Petri net, also called a marking, is a mapping of the places on the natural numbers. With an implicit order over the places, a marking is simply a vector of natural numbers. Formally, a Petri net is given as a finite set of actions, and an action a is a pair $(\bar{u}, \bar{v})$ of markings denoting respectively the pre-condition $\bar{u}$ and the post-condition $\bar{v}$ of a.



$$
\begin{array}{lcl}
a_1 & = & (\,(1,1,0,0,0)\,,\ (1,0,1,0,0)\,)\\
a_2 & = & (\,(1,0,0,0,0)\,,\ (0,0,0,1,0)\,)\\
a_3 & = & (\,(0,0,0,1,0)\,,\ (1,0,0,0,1)\,)\\
a_4 & = & (\,(0,0,1,1,0)\,,\ (0,2,0,1,0)\,)
\end{array}
$$

Fig. 6: A Petri net with places $\{p_1, p_2, p_3, p_4, p_5\}$ and actions $\{a_1, a_2, a_3, a_4\}$.

The semantics of an action $a = (\bar{u}, \bar{v})$ is defined thanks to a binary relation $\xrightarrow{a}$ over the markings as follows:

$$\bar{x} \xrightarrow{a} \bar{y} \ \text{ iff } \ \bar{x} \geq \bar{u} \ \wedge \ \bar{y} = \bar{x} - \bar{u} + \bar{v}$$

The reachability problem takes as input a Petri net and a pair $(\bar{x}, \bar{y})$ of markings, and decides if there exists a sequence $a_1, \ldots, a_k$ of actions (an action can occur many times) such that:

$$\bar{x} \xrightarrow{a_1} \cdots \xrightarrow{a_k} \bar{y}$$

In this case $\bar{y}$ is said to be reachable from $\bar{x}$, and the set of markings reachable from $\bar{x}$ is called the reachability set from $\bar{x}$.

Example 7 The Petri net depicted Fig. 6 is equipped with the initial marking $\bar{x} = (1, 1, 0, 0, 0)$. From [39,40], we deduce that for every non reachable marking $\bar{y}$, there exists a Presburger formula denoting an inductive invariant that contains the initial marking and not the marking $\bar{y}$. This Presburger formula must depend on $\bar{y}$. In fact, if there exists a Presburger formula denoting an inductive invariant that contains the initial marking and only reachable markings, then this formula denotes the reachability set of the Petri net. However, the Petri net depicted in that figure is a classical example of a Petri net with a reachability set that is not definable in Presburger arithmetic (see [35] for more details).

9.2 CEGAR for Petri Nets

The CEGAR approach (Counter Example Guided Abstract Refinement) [25,28] provides a general framework for automatically computing inductive invariants. In this approach, a finite set of formulas in a decidable logic, called predicates, are used to transform a concrete system into an abstract one. Informally, the abstract system is a finite graph; states are labeled by Boolean combinations of predicates; actions are labeled by actions of the Petri net in such a way the finite graph simulates the Petri net. For Petri nets, Presburger arithmetic is a good candidate for denoting predicates. In fact, this logic is sufficient for denoting witnesses of non-reachability as explained in related work (see Section 2).

With this approach, and the predicates $\{p_1 = 0, p_1 = 1, p_4 = 0, p_4 = 1\}$, the Petri net depicted Fig. 6 is abstracted away into the finite graph depicted in Fig. 7.

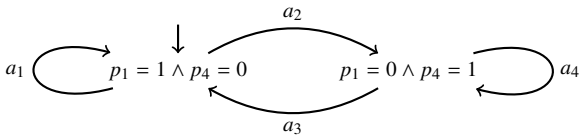


Fig. 7: An abstraction of a Petri net.

This graph provides a simple way for proving that a marking is not reachable from the initial marking $\bar{x} = (1, 1, 0, 0, 0)$. In fact, the formula obtained as the disjunction of all the formulas labeling states denotes an inductive invariant. Hence, a marking that does not satisfies the following formula is not reachable:

$$(p_1 = 1 \wedge p_4 = 0) \vee (p_1 = 0 \wedge p_4 = 1)$$

Conversely, if a marking $\bar{y}$ satisfies this formula, we cannot conclude that $\bar{y}$ is reachable. However, in this case there exists a path from the initial state to a state labeled by a formula satisfied by $\bar{y}$. Such a path can be automatically computed with classical graph exploration algorithms. This path is labeled by a sequence of actions $a_1, \ldots a_k$ and one can simply check if the following property holds:

$$\bar{x} \xrightarrow{a_1} \cdots \xrightarrow{a_k} \bar{y}$$

In the positive case, we deduce that $\bar{y}$ is reachable. In the negative case, it means that the abstraction is too coarse. In order to improve the preciseness of the abstraction, new predicates must be computed. These predicates are obtained by observing that the following formula is unsatisfiable where $\phi_j[\bar{s}_{j-1}, \bar{s}_j]$ is the formula $\bar{s}_{j-1} \geq \bar{u}_j \wedge \bar{s}_j + \bar{u}_j = \bar{s}_{j-1} + \bar{v}_j$ encoding the semantics of the action $a_j = (\bar{u}_j, \bar{v}_j)$:

$$\begin{aligned}&(\bar{s}_0 = \bar{x} \wedge \phi_1[\bar{s}_0, \bar{s}_1] \wedge \ldots \wedge \phi_j[\bar{s}_{j-1}, \bar{s}_j])\\ \wedge&(\phi_{j+1}[\bar{s}_j, \bar{s}_{j+1}] \wedge \ldots \wedge \phi_k[\bar{s}_{k-1}, \bar{s}_k] \wedge \bar{s}_k = \bar{y})\end{aligned}$$

Since the formula is unsatisfiable, an interpolant $I_j[\bar{s}_j]$ for the previous interpolation problem can be computed. This interpolant is then added to the set of predicates, and we restart the analysis of the Petri net with the new set of predicates.

The convergence of the CEGAR approach relies on the quality of the computed predicates. Unfortunately, a computation of interpolants as previously explained provides low quality predicates for Petri nets. The problem comes from the fact that if the previously given formula is unsatisfiable, there exists a very simple explanation of this property that only relies on the particular value of a components of $\bar{s}_j$. In practice, the predicates computed this way are only Boolean combinations of formulas of the form $p_n = c$ where c is some natural number. The CEGAR loop will diverge on some simple Petri nets as shown in Example 8.

Example 8 The Petri net depicted Fig. 8 is initialized with the zero marking $(0, 0)$. The reachability set is the set of markings satisfying $p_1 = p_2$. The only inductive invariant that contains the initial marking and is a Boolean combination of formulas $p_1 = c$ or $p_2 = c$ is the formula *true*. In particular for this Petri net, the classical CEGAR approach as presented in this section will diverge for proving that $(1, 0)$ is not reachable.

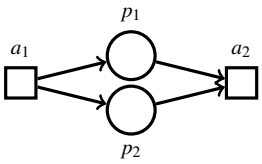


Fig. 8: Another Petri net.

9.3 Interpolation Abstraction for Generating Predicates

We apply the CEGAR loop exploration as previously presented, but interpolants are computed from sequences of actions $a_1, \ldots, a_k$ only after application of interpolation abstraction. This serves the purpose of improving the quality of predicates generated by interpolation, so that divergence of CEGAR can be avoided in many (though not all) cases. Our approach to compute interpolation abstractions is based on different heuristics combining linear algebra and acceleration techniques [22]. In the sequel, we present three different kinds of interpolation abstractions.

Global-orthogonal-space heuristic (ABS (1)).
The computation of place invariants is a classical way for efficiently computing invariants of Petri nets. Place invariants are obtained by observing that if a vector $\bar{t}$ is orthogonal to $\bar{v} - \bar{u}$ for every action $a = (\bar{u}, \bar{v})$ of the Petri net, then $\bar{t}$ is orthogonal to $\bar{y} - \bar{x}$ for every marking $\bar{y}$ reachable from $\bar{x}$. That means the dot product of $\bar{t}$ with any reachable marking is a constant. Our first heuristic is based on the observation that orthogonal vectors t are suitable templates to be used in combination with term or inequality interpolation abstractions (Sect. 5.1 and 5.2). We first compute a basis of the vector space ortogonal to all vectors $\bar{v}_j - \bar{u}_j$ where $a_j = (\bar{u}_j, \bar{v}_j)$. This basis is then completed as an orthogonal basis B of the whole vector space generated by the markings. Such a computation is performed with Gauss elimination in polynomial time.

We then define an abstraction lattice using the powerset lattice for B (Def. 9), with each node in the lattice mapping to an inequality interpolation abstraction for some subset of B. The abstraction lattice is equipped with a *cost* function (as in Sect. 7.3) that maps orthogonal vectors $\bar{t}$ to a small cost, and all other basis vectors to a large cost. As a result, the search procedures from Sect. 7 are able to systematically search for interpolation abstractions, and consequently interpolants, that are defined using orthogonal vectors; such interpolants are likely invariant under all or many actions of a Petri net.

Acceleration of individual recurring actions (ABS (2)).
Acceleration techniques compute reachability sets thanks to the exact effect of iterating some sequences of actions. For instance, let us consider an action $a = (\bar{u}, \bar{v})$, and observe that for every natural number $n \geq 1$, we have $\bar{x} \xrightarrow{a^n} \bar{y}$ if and only if $\bar{x} \geq \bar{u}$, $\bar{y} \geq \bar{v}$, and $\bar{y} + n.\bar{u} = \bar{x} + n.\bar{v}$. Our second heuristic is based on acceleration. Basically, rather than computing interpolants directly from a sequence of actions $a_1, \ldots, a_k$, we compute interpolants $I_j[s_j]$ using the following formula, where $\phi_j^{acc}[\bar{s}_{j-1}, n_j, \bar{s}_j]$ is the formula $\bar{s}_{j-1} \geq \bar{u}_j \wedge \bar{s}_j \geq \bar{v}_j \wedge \bar{s}_j + n_j.\bar{u}_j = \bar{s}_{j-1} + n_j.\bar{v}_j$ encoding the effect of iterating n_j times the action $a_j = (\bar{u}_j, \bar{v}_j)$.

$$(\bar{s}_0 = \bar{x} \wedge \phi_1^{acc}[\bar{s}_0, n_1, \bar{s}_1] \wedge \ldots \wedge \phi_j^{acc}[\bar{s}_{j-1}, n_j, \bar{s}_j])$$
$$\wedge(\phi_{j+1}^{acc}[\bar{s}_j, n_{j+1}, \bar{s}_{j+1}] \wedge \ldots \wedge \phi_k^{acc}[\bar{s}_{k-1}, n_k, \bar{s}_k] \wedge \bar{s}_k = \bar{y})$$

Note that $\exists n_j \geq 1.\ \phi_j^{acc}[\bar{s}_{j-1}, n_j, \bar{s}_j]$ is an over-approximation of $\phi_j[\bar{s}_{j-1}, \bar{s}_j]$, and can in fact be mapped to an inequality interpolation abstraction by means of quantifier elimination. As before, costs can be used to steer interpolant exploration towards interpolants that are invariant under recurrence of the accelerated action.

Detection of increasing sequences (ABS (3)).
In our last heuristics, we abstract away the sequence $a_1, \ldots, a_k$ of actions as a multiset. This abstraction basically extracts the Parikh image by counting the number of times an action occurs. Informally, thanks to linear algebra methods, sub-multisets of actions are computed in such a way that the effect of these actions is a non-negative vector. More formally, we consider for each action $a = (\bar{u}_a, \bar{v}_a)$ a natural number n_a in such a way that the following vector $\bar{v}$ satisfies $\bar{v} \geq \bar{0}$:

$$\bar{v} = \sum_{a \in A} n_a(\bar{v}_a - \bar{u}_a)$$

The computation of vectors $\bar{v}$ satisfying $\bar{v} \geq \bar{0}$ is motivated by the framework of acceleration as previously mentioned. In fact, an action $a = (\bar{u}, \bar{v})$ with a non-negative effect $\bar{v} - \bar{u}$ can be iterated an arbitrary number of times; the same can be the case for non-negative vectors $\bar{v}$ combining the effect of several actions.

Example 9 The considered vectors $\bar{v}$ may not correspond to any sequence of actions that is iterable from any reachable marking. For instance, let us consider the Petri net depicted in Fig. 6. The abstraction of this Petri net depicted in Fig. 7 shows that the sequence a_1a_4 cannot be iterated (the actions a_2 and a_3 must be taken between a_1 and a_4). However, the effect of a_1a_4 is equal to $(0, 1, 0, 0, 0)$ which is a non-negative vector, and logically the pair a_1a_4 of actions has a similar effect as an ordinary loop.

Once a non-negative vector $\bar{v}$ has been extracted from $a_1, \ldots, a_k$, a lattice of inequality interpolation abstractions is derived by computing a basis B' of the vector space orthogonal to $\bar{v}$, and a basis $B = B' \cup \{\bar{v}\}$ of the complete space. The vectors in B are then used as templates, with a *cost* function assigning large cost to $\bar{v}$, and small cost to the vectors in B'.

9.4 Experiments

In order to evaluate the efficacy of the different interpolation abstractions, we implemented a Petri net checker on the basis of the model checker Eldarica (Sect. 8.2) and integrated the three forms of abstraction defined in the previous section. Experiments were done using a set of (bounded and unbounded) Petri net benchmarks taken from the literature.[8]

The results are given in Table 2, in terms of runtime and the required number of CEGAR iterations. As can be seen, Eldarica without interpolation abstraction performs poorly on Petri nets, and times out in many cases. The three interpolation abstractions show complementary performance, and each of our benchmarks could be solved using at least one of the heuristics. A combination of the interpolation abstractions (**ABS-all**) is also able to solve all benchmarks, although not always with the best runtime.

Finally, we compared to the acceleration-based model checker Fast [3]. Fast checks reachability queries by first computing a closed-form representation of the complete reachability set, and therefore has the same runtime for reachable as for unreachable cases. Fast is able to solve all bounded Petri nets in very short time, but times out for a number of the unbounded ones. In particular, Fast fails for the "Exponential" example, which is similar to Fig. 6 and has a reachability set that cannot be defined in Presburger arithmetic.

[8] Benchmarks and implementation on `http://www.philipp.ruemmer.org/eldarica-p.shtml`

Benchmark		Eldarica		ABS (1)		ABS (2)		ABS (3)		ABS-all		Fast
		N	sec	N	sec	N	sec	N	sec	N	sec	sec
Bounded Petri nets												
Basic ME	U	3	1.3	3	1.55	3	1.3	3	1.3	3	1.7	**<1**
IFIP	U	12	2.3	2	1.7	12	4.3	10	4.6	2	1.8	**<1**
L6000	U	*	*	17	16.5	8	4.7	*	*	3	4.0	**<1**
Long 1	U	*	*	1	1.2	7	7.1	*	*	1	1.2	**<1**
Long 2	U	*	*	1	1.4	10	11.1	13	15.4	1	1.4	**<1**
Long 3	U	*	*	*	*	10	11.5	8	8.2	11	19.2	**<1**
Long 4	U	*	*	1	2.8	9	11.2	103	79.6	1	3.0	**<1**
Manufacturing 3	U	*	*	323	802	441	2635	675	1946	354	1588	**2.4**
Manufacturing 9	R	*	*	232	801	264	632	560	3053	295	1515	**10.8**
Unbounded Petri nets												
Alternating bit prot.	R	64	14.8	16	10.5	44	17.5	35	15.2	16	14.7	**4.5**
FMS	R	25	**20.5**	23	28.4	25	27.3	17	24.7	23	32.4	98.4
”	U	18	9.8	2	7.0	13	17.6	18	10.7	2	**6.7**	37.4
FinkelKM	R	16	5.8	15	8.9	16	11.6	17	11.6	15	22.7	**5.7**
”	U	14	5.7	3	**2.4**	6	6.5	7	3.4	3	2.5	5.7
Finkel Counterex.	R	12	2.3	10	3.5	12	2.3	12	2.6	10	3.6	**<1**
Kanban	R	28	**33.3**	19	35.8	29	70.0	22	41.5	25	67.3	*
”	U	*	*	1	3.9	*	*	*	*	1	**3.8**	*
Mesh 2x2	R	75	**52.3**	64	82.9	60	56.6	68	102	65	105	97
”	U	186	170	18	**33.7**	*	*	*	*	18	37.8	97
Multipool	U	56	423	1	5.4	*	*	*	*	1	**5.0**	*
Pingpong	U	3	1.4	2	1.5	2	1.4	2	1.3	2	1.5	**<1**
PNCSA Cover	R	32	15.0	17	16.5	32	**14.5**	26	16.4	17	17.7	*
Exponential	U	*	*	8	3.9	8	**3.4**	6	5.1	5	5.2	*
Language inclusion	U	*	*	*	*	5	3.7	2	1.7	6	9.0	**<1**

Table 2: Comparison of tools for checking reachability in bounded and unbounded Petri nets, on benchmarks taken from the literature. **Eldarica** is the standard CEGAR engine of the Eldarica model checker [33]. **ABS (1)** is the global-orthogonal-space heuristic, **ABS (2)** accelerates individual actions, **ABS (3)** detects increasing sequences, **ABS-all** combines all abstraction methods. **Fast** is the acceleration-based model checker [3]. For each benchmark, “U” denotes that the considered configuration is unreachable, while “R” represents reachable configurations. Items with “*” indicate a timeout (set to 1 hour). Experiments with Eldarica were done on an Intel Core i5 2-core machine with 3.2GHz; Fast was run on an Intel Core i7 2-core machine with 1.7GHz.

10 Conclusion

We have presented a semantic solver-independent framework for guiding theorem provers towards high-quality interpolants. Our method is simple to implement, but can improve the performance of model checkers significantly. The framework makes it easy to augment interpolation procedures with domain-specific techniques to construct useful invariants, and we therefore expect it to be helpful in many domains where standard unguided interpolation methods struggle to generate good solutions.

Various directions of future work are planned, among others: (i) we want to explore further forms of interpolation abstractions, in particular for computing quantified interpolants and for handling infinite abstraction lattices; (ii) it is planned to gather more experiences with interpolation abstraction for programs with arrays and heap, which are notoriously difficult for standard interpolation methods; (iii) since our approach is clearly related to the theory of abstract interpretation, we plan to investigate whether further methods from abstract interpretation can be carried over to our framework.

Acknowledgements We thank Hossein Hojjat and Viktor Kuncak for discussions, and for assistance with the implementation in Eldarica. We are also grateful for helpful comments from the referees.

References

1. Albarghouthi, A., McMillan, K.L.: Beautiful interpolants. In: CAV, pp. 313–329 (2013)
2. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy abstraction with interpolants for arrays. In: LPAR (2012)
3. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: acceleration from theory to practice. International Journal on Software Tools for Technology Transfer (STTT) **10**(5), 401–424 (2008)
4. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: VMCAI. Springer (2007)
5. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: G. Gopalakrishnan, S. Qadeer (eds.) CAV, *LNCS*, vol. 6806, pp. 184–190. Springer (2011). DOI 10.1007/978-3-642-22110-1_16
6. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: R. Bloem, N. Sharygina (eds.) FMCAD, pp. 189–197. IEEE (2010)
7. Beyer, D., Zufferey, D., Majumdar, R.: CSIsat: Interpolation for LA+EUF. In: CAV, *LNCS*, vol. 5123, pp. 304–308. Springer (2008)
8. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. In: CAV, pp. 517–531 (2006)
9. Bozga, M., Habermehl, P., Iosif, R., Konečný, F., Vojnar, T.: Automatic verification of integer array programs. In: CAV, pp. 157–172 (2009)
10. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An interpolating sequent calculus for quantifier-free Presburger arithmetic. In: Proceedings, IJCAR, LNCS. Springer (2010)
11. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: Beyond quantifier-free interpolation in extensions of Presburger arithmetic. In: VMCAI, LNCS. Springer (2011)
12. Caniart, N., Fleury, E., Leroux, J., Zeitoun, M.: Accelerating interpolation-based model-checking. In: TACAS, pp. 428–442 (2008)
13. Chaudhuri, S., Farzan, A., Kincaid, Z.: Consistency analysis of decision-making programs. In: S. Jagannathan, P. Sewell (eds.) POPL, pp. 555–568. ACM (2014). DOI 10.1145/2535838.2535858
14. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: E. Ábrahám, K. Havelund (eds.) TACAS, *LNCS*, vol. 8413, pp. 46–61. Springer (2014)
15. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. The Journal of Symbolic Logic **22**(3) (1957)
16. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order (2. ed.). Cambridge University Press (2002)
17. Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: A.L. Hosking, P.T. Eugster, C.V. Lopes (eds.) OOPSLA, pp. 443–456. ACM (2013)
18. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: VMCAI, pp. 129–145 (2010)
19. Esparza, J., Nielsen, M.: Decidability issues for Petri nets - a survey. Bulletin of the European Association for Theoretical Computer Science **52**, 245–262 (1994)
20. Felsing, D., Grebing, S., Klebanov, V., Ulbrich, M., Rümmer, P.: Automating regression verification. In: ASE (2014)
21. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: FME, pp. 500–517 (2001)
22. Fribourg, L.: Petri nets, flat languages and linear arithmetic. In: M. Alpuente (ed.) Proc. of WFLP'2000, pp. 344–365 (2000)
23. Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. CoRR **abs/1011.0551** (2010)
24. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)
25. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: CAV, pp. 72–83 (1997)
26. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI (2012)
27. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press (2009)

28. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: 31st POPL (2004)
29. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, pp. 232–244. ACM (2004)
30. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: SAT, pp. 157–171 (2012)
31. Hoder, K., Kovács, L., Voronkov, A.: Playing in the grey area of proofs. In: POPL, pp. 259–272 (2012)
32. Hojjat, H., Iosif, R., Konecný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: ATVA, pp. 187–202 (2012)
33. Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A verification toolkit for numerical transition systems - tool paper. In: FM, pp. 247–251 (2012)
34. Hojjat, H., Rümmer, P., Subotic, P., Yi, W.: Horn clauses for communicating timed systems. In: Workshop on Horn Clauses for Verification and Synthesis (2014)
35. Hopcroft, J.E., Pansiot, J.J.: On the reachability problem for 5-dimensional vector addition systems. Theoritical Computer Science **8**, 135–159 (1979)
36. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: TACAS, pp. 459–473 (2006)
37. Kosaraju, S.R.: Decidability of reachability in vector addition systems (preliminary version). In: Proc. of STOC'82, pp. 267–281. ACM (1982)
38. Lambert, J.L.: A structure to decide reachability in Petri nets. Theoretical Computer Science **99**(1), 79–104 (1992)
39. Leroux, J.: The general vector addition system reachability problem by Presburger inductive invariants. In: Proc. of LICS 2009, pp. 4–13. IEEE Computer Society (2009)
40. Leroux, J.: Vector addition system reachability problem: a short self-contained proof. In: Proc. of POPL'11, POPL '11, pp. 307–316. ACM (2011)
41. Marques-Silva, J., Janota, M., Belov, A.: Minimal sets over monotone predicates in boolean formulae. In: CAV, pp. 592–607 (2013)
42. Mayr, E.W.: An algorithm for the general Petri net reachability problem. In: Proc. of STOC'81, pp. 238–246. ACM (1981)
43. McMillan, K.L.: Lazy abstraction with interpolants. In: CAV (2006)
44. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: TACAS, pp. 413–427 (2008)
45. Popeea, C., Chin, W.: Dual analysis for proving safety and finding bugs. Sci. Comput. Program. **78**(4), 390–411 (2013). DOI 10.1016/j.scico.2012.07.004. URL `http://dx.doi.org/10.1016/j.scico.2012.07.004`
46. Rollini, S., Bruttomesso, R., Sharygina, N.: An efficient and flexible approach to resolution proof reduction. In: HVC, pp. 182–196 (2010)
47. Rollini, S.F., Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Periplo: A framework for producing effective interpolants in sat-based software verification. In: K.L. McMillan, A. Middeldorp, A. Voronkov (eds.) LPAR, *LNCS*, vol. 8312, pp. 683–693. Springer (2013). DOI 10.1007/978-3-642-45221-5
48. Rollini, S.F., Sery, O., Sharygina, N.: Leveraging interpolant strength in model checking. In: CAV, pp. 193–209 (2012)
49. Rümmer, P., Hojjat, H., Kuncak, V.: Classifying and solving Horn clauses for verification. In: E. Cohen, A. Rybalchenko (eds.) Verified Software: Theories, Tools, Experiments - 5th International Conference (VSTTE), *LNCS*, vol. 8164, pp. 1–21. Springer (2013)
50. Rümmer, P., Hojjat, H., Kuncak, V.: Disjunctive interpolants for Horn-clause verification. In: Computer Aided Verification (CAV), *LNCS*, vol. 8044, pp. 347–363. Springer (2013)
51. Rümmer, P., Subotić, P.: Exploring interpolants. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 69–76. IEEE (2013)
52. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint solving for interpolation. In: Proceedings, VMCAI, *LNCS*, vol. 4349, pp. 346–362. Springer (2007)
53. Seghir, M.N.: A lightweight approach for loop summarization. In: ATVA, pp. 351–365 (2011)
54. Smrcka, A., Vojnar, T.: Verifying parametrised hardware designs via counter automata. In: Haifa Verification Conference, pp. 51–68 (2007)
55. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: PLDI, pp. 223–234 (2009)
56. Tonetta, S.: Abstract model checking without computing the abstraction. In: A. Cavalcanti, D. Dams (eds.) FM, *LNCS*, vol. 5850, pp. 89–105. Springer (2009)
57. Totla, N., Wies, T.: Complete instantiation-based interpolation. In: R. Giacobazzi, R. Cousot (eds.) POPL, pp. 537–548. ACM (2013)