Exploring Interpolants

Philipp Rümmer, Pavle Subotić
Department of Information Technology, Uppsala University, Sweden

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 exploited in the context of software model checking.**

I. 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* [1], a logical tool to extract concise explanations for the infeasibility (safety) of specific paths in a program. To ensure rapid convergence, model checkers rely on theorem provers to find suitable interpolants, or interpolants containing the right predicates, 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. II 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 [2], which uses recursion-free Horn clauses (a generalisation of Craig interpolation) to construct abstractions [3], [4]. Our experiments show that interpolation abstraction can prevent divergence of the model checker in cases that are often considered challenging.

A. Introductory Example

We consider an example inspired by the program discussed in the introduction of [5]. 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:

$$i_0 \doteq 0 \wedge x_0 \doteq j \wedge i_0 < 50 \wedge i_1 \doteq i_0{+}1 \wedge x_1 \doteq x_0{+}1 \qquad (1)$$
$$\wedge\, i_1 \geq 50 \wedge j \doteq 0 \wedge x_1 < 50 \qquad (2)$$

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 $(1) \wedge (2)$, 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 $(1) \to I$ and $(2) \to \neg I$, 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 $(1) \wedge (2)$ 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 paper 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:

$$\begin{aligned}
&(i_0 \doteq 0 \wedge x_0 \doteq j' \wedge i_0 < 50 \wedge \\
&\quad i_1' \doteq i_0{+}1 \wedge x_1' \doteq x_0{+}1 \wedge \boxed{x_1' - i_1' \doteq x_1 - i_1 \wedge j' \doteq j}\,) \\
&\wedge (\,\boxed{x_1 - i_1 \doteq x_1'' - i_1'' \wedge j \doteq j''} \wedge i_1'' \geq 50 \wedge j'' \doteq 0 \wedge x_1'' < 50)
\end{aligned}$$

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 Sect. VII illustrate the generality of this simple kind of template). Template-generated abstractions represent the most important class of interpolation abstractions considered in this paper (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. To illustrate this, observe that the templates $\{x_1 - i_1, i_1\}$ would generate *the same* interpolation abstraction as $\{x_1, i_1\}$; this is because the values of $x_1 - i_1, i_1$ uniquely determine the value of x_1, i_1, and vice versa.

B. Contributions and Organisation of this Paper

- The framework of *interpolant abstractions* (Sect. IV);
- A catalogue of interpolation abstractions, in particular interpolation abstractions generated from *template terms* and *template predicates* (Sect. V);
- Algorithms to explore *lattices of interpolation abstractions,* in order to compute a range of interpolants for a given interpolation problem (Sect. VI);
- An experimental evaluation (Sect. VII).

II. Related Work

Syntactic restrictions of considered interpolants [5], [6], 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 paper. 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 [7], [8]: 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. I-A 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 [9], [10], applied to the same propositional resolution proof. To the best of our knowledge, no conclusive results are available relating interpolant strength with model checking performance. 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.

In a similar fashion, proofs and interpolants can be **minimised** by means of proof transformations [11], [12]. 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 [13], [14], [15]. 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 programs. 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 [16], [17], [18], 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.

III. Preliminaries

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., [19]. The main logics considered in this paper 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 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; $\phi[\bar{s}']$ ($t[\bar{s}']$) is the variant of $\phi[\bar{s}]$ ($t[\bar{s}]$) 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 $B[\bar{s}, \bar{s}_B] \Rightarrow \neg I[\bar{s}]$; the existence of an interpolant 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; however, many of the results hold more generally.

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 $x_1, \ldots, x_n$.

2) Statelessness: Some of the results presented in this paper 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); other logics, for instance full FOL, modal logics, or separation logic can be made stateless by enriching its vocabulary. Statelessness is important in this paper, since we use the concept of *renaming* of symbols to ensure independence of formulae.

3) 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* of a if $a \sqsubseteq b$ but $a \neq b$, and *immediate successor* if in addition there is no $c \in D \backslash \{a, b\}$ with $a \sqsubseteq c \sqsubseteq b$. 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 $\bigsqcup\{a, b\}$ and $\bigsqcap\{a, b\}$ 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 $\bigsqcup\{a, b\} \in M$ and $\bigsqcap\{a, b\} \in M$ for all $a, b \in M$.

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 $f(x) \sqsubseteq_2 f(y)$.

IV. Interpolation Abstractions

This section defines the general concept of interpolation abstractions, and derives basic properties:

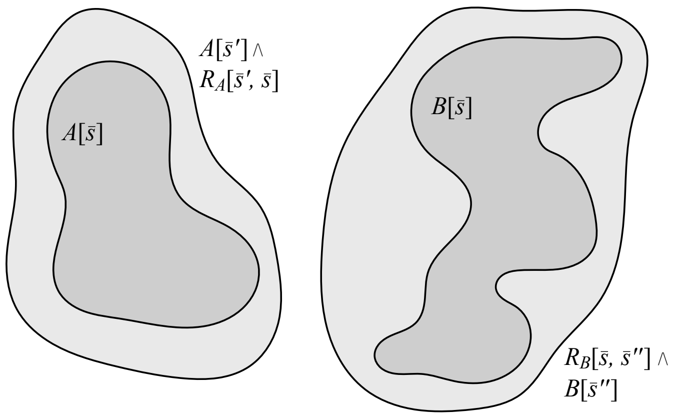

Fig. 1. Illustration of interpolation abstraction, assuming that only common non-logical symbols exist. Both concrete and abstract problem are solvable.

Definition 2 (Interpolation abstraction) Suppose $A[\bar{s}_A, \bar{s}]$ and $B[\bar{s}, \bar{s}_B]$ are formulae over disjoint lists $\bar{s}_A, \bar{s}, \bar{s}_B$ of non-logical symbols, and $\bar{s}'$ and $\bar{s}''$ fresh copies of $\bar{s}$. An *interpolation 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 $Id[\bar{s}, \bar{s}''] \Rightarrow R_B[\bar{s}, \bar{s}'']$). We call $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ the *concrete interpolation problem,* and

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

the *abstract interpolation problem* for $A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B]$ and $(R_A[\bar{s}', \bar{s}], R_B[\bar{s}, \bar{s}''])$.

Assuming that the concrete interpolation problem is solvable, we call an interpolation abstraction *feasible* if also the abstract interpolation problem is solvable, and *infeasible* otherwise.

The common symbols of the interpolation problem in Sect. I-A are $\bar{s} = \langle x_1, i_1, j \rangle$, and the interpolation abstraction is defined by $R_A = (x_1' - i_1' \doteq x_1 - i_1 \wedge j' \doteq j)$ and $R_B = (x_1 - i_1 \doteq x_1'' - i_1'' \wedge j \doteq j'')$. A further 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 $B[\bar{s}]$. Since $R_A[\bar{s}, \bar{s}]$ and $R_B[\bar{s}, \bar{s}]$ are valid, the solution set of $A[\bar{s}_A, \bar{s}'] \wedge R_A[\bar{s}', \bar{s}]$ represents an *over-approximation* of $A[\bar{s}]$; similarly for $B[\bar{s}]$ and $R_B[\bar{s}, \bar{s}'']$. This ensures the soundness of computed abstract interpolants. In Fig. 1, despite over-approximation, the abstract interpolation problem is solvable, which means that the interpolation abstraction is feasible.

Lemma 3 (Soundness) Every interpolant of the abstract interpolation problem is also an interpolant of the concrete interpolation problem (but in general not vice versa).

Interpolation abstractions can be used to guide interpolation engines, by restricting the space $Inter(A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B])$ of interpolants satisfying an interpolation problem. For this, recall that the set $Inter(A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B])/\!\equiv$ of interpolant classes (modulo logical equivalence) is closed under conjunctions (meet) and disjunctions (join), so that $(Inter(A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B])/\!\equiv, \Rightarrow)$ is a lattice. Fig. 2 shows the

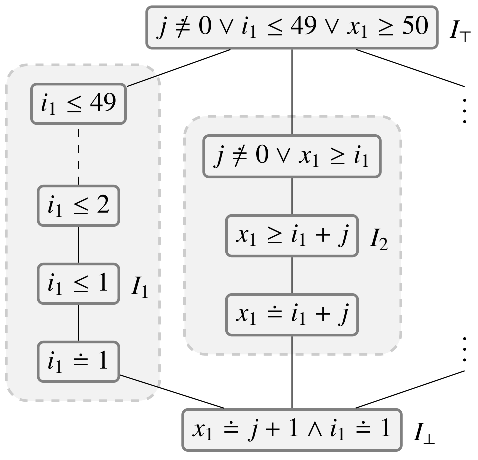

Fig. 2. Parts of the interpolant lattice for the example in Sect. I-A (up to equivalence). The dashed boxes represent the sub-lattices for the abstraction induced by the template terms $\{i_1\}$ (left) and $\{x_1 - i_1, j\}$ (right).

interpolant lattice for the example in Sect. I-A; this lattice has a strongest concrete interpolant $I_\bot$ and a weakest concrete interpolant $I_\top$. In general, the interpolant lattice might be incomplete and not contain such elements.

For a feasible abstraction, the lattice of abstract interpolants

$$(\mathit{Inter}(A[\bar{s}_A, \bar{s}'] \land R_A[\bar{s}', \bar{s}],\ R_B[\bar{s}, \bar{s}''] \land B[\bar{s}'', \bar{s}_B])/\!\equiv,\ \Rightarrow)$$

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 relation R_A in an interpolation abstraction constrains the lattice of abstract interpolants from below, the relation R_B from above.

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

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

used in Sect. I-A 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 4 (Completeness) Suppose $A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B]$ 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 in the same logic) such that every abstract interpolant is equivalent to $I[\bar{s}]$.

V. A Catalogue of Interpolation Abstractions

This section introduces a range of practically relevant interpolation abstractions, mainly defined in terms of *templates* as illustrated in Sect. I-A. 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. I-A) 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.

A. 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. I-A. 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. 7 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. I-A, 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 5 (Term interpolation abstraction) Suppose that $A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B]$ is an interpolation problem, and $T = \{t_1[\bar{s}], \ldots, t_n[\bar{s}]\}$ a finite set of ground terms. The interpolation abstraction $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ defined by

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

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 6 (Solvability) Suppose $A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B]$ 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 $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ 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 $A[\bar{s}_A, \bar{s}] \land B[\bar{s}, \bar{s}_B]$.

Example 1 Consider the interpolation abstraction used in Sect. I-A, 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. 6, 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. IV) that interpolation can only be restricted *up to logical equivalence:*

Lemma 7 (Interpolant space) Suppose the abstract interpolation problem for $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ 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.

Example 2 Again, consider Sect. I-A, 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 $j \not\doteq 0 \vee x_1 \geq i_1$.

B. Finite Predicate Interpolation Abstractions

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 [20], [21], but still leaves the use of templates entirely to the theorem prover.

Definition 8 (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. $(R_A^{Pred}[\bar{s}', \bar{s}], R_B^{Pred}[\bar{s}, \bar{s}''])$ defined by

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

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*. 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 9 (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*.

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 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. 7):

Lemma 10 (Interpolant space) Suppose the abstract interpolation problem for $(R_A^{Pred}[\bar{s}', \bar{s}], R_B^{Pred}[\bar{s}, \bar{s}''])$ 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.

C. 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 3 Suppose 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:

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

An obvious interpolant is the formula $I_1 = (H(b, g) \not\doteq null)$. Based on domain-specific knowledge, 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 11 (Schematic term abstraction) Suppose an interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, and a finite set $T = \{t_1[\bar{s}, \bar{x}_1], \ldots, t_n[\bar{s}, \bar{x}_1]\}$ of terms with free variables $\bar{x}_1, \ldots, \bar{x}_n$. The interpolation abstraction $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ defined by

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

is called *schematic term interpolation abstraction* over T.

Note that schematic term interpolation abstractions reduce to ordinary term interpolation abstractions (as in Def. 5) 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 12 (Solvability in EUF) Suppose $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is an interpolation problem in EUF, $T = \{t_1[\bar{s}, \bar{x}_1], \ldots, t_n[\bar{s}, \bar{x}_1]\}$ 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 $\bar{x}_i$.

VI. Exploration of Interpolants

In practice, given an interpolation problem, we want to compute a whole *range* of interpolants, or alternatively find an interpolant that is optimal with respect to some objective. For instance, in the example in Sect. I-A, we consider interpolant I_2 constructed using templates $\{x_1 - i_1, j\}$ as "better" than interpolant I_1 for the template i_1. To formalise this concept of *interpolant exploration* we arrange families of interpolation abstractions as *abstraction lattices,* and present search algorithms on such lattices. Abstraction lattices are equipped with a monotonic mapping μ to abstractions (R_A, R_B), ordered by component-wise implication. The following paragraphs focus on the case of *finite* abstraction lattices; the handling of infinite (parametric) abstraction lattices is planned as future work.

Definition 13 (Abstraction lattice) Suppose an interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$. 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 $(R_A[\bar{s}', \bar{s}], R_B[\bar{s}, \bar{s}''])$ with the property that $\mu(\perp) = (Id[\bar{s}', \bar{s}], Id[\bar{s}, \bar{s}''])$.

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 $\mu(\perp) = (Id[\bar{s}', \bar{s}], Id[\bar{s}, \bar{s}''])$.

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 as *abstraction frontier*.

A. 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 both to term and predicate templates. Another useful construction is to form the *product* of two lattices, defining the mapping μ as the conjunction (or alternatively disjunction) of the individual mappings μ_1, μ_2.

Example 4 An abstraction lattice for the example in Sect. I-A 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. 5. 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 $(Id[\bar{s}', \bar{s}], Id[\bar{s}, \bar{s}''])$, 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 $\perp$) 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 $I_3 = (j \not\equiv 0 \vee i_1 \leq 49)$.

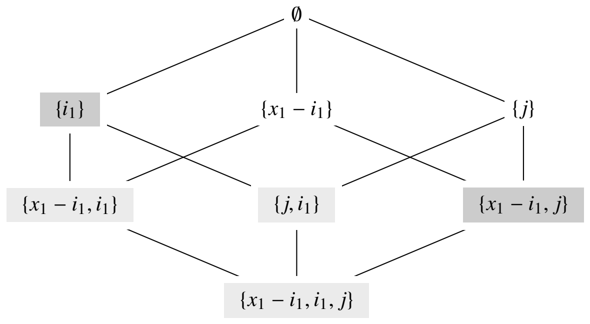

Fig. 3. The abstraction lattice for the running example. The light gray shaded elements are feasible, the dark gray ones maximal feasible.

B. Computation of Abstraction Frontiers

We present an algorithm to compute abstraction frontiers of finite abstraction lattices. The search is described in Algorithms 1 and 2. Algorithm 1 describes the top-level procedure for finding minimal elements in an abstraction lattice. Initially we check if the $\perp$ element is infeasible (line 1). If this is the case, then the concrete interpolation problem is not solvable and we return an empty abstraction frontier. Otherwise, we initialise the frontier with a maximal feasible element (line 4), which is found by the *maximise* function (described in Algorithm 2). Next, in line 5 we check whether a feasible element can be found that is incomparable to all frontier elements found so far; efficient methods for computing such incomparable elements can be defined based on the shape of the chosen abstraction lattice, and are not shown here. As long as incomparable elements can be found, we compute further maximal feasible elements and add them to the frontier.

In Algorithm 2 we describe the procedure for finding a maximal feasible element *mfe* with the property that *elem* $\sqsubseteq$ *mfe*. In each iteration of the maximisation loop, it is checked whether

Algorithm 1: Exploration algorithm

```
Input: Interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$,
       abstraction lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$
Result: Set of maximal feasible interpolation abstractions
1 if $\bot$ is infeasible then
2     return $\emptyset$;
3 end
4 $Frontier \leftarrow \{maximise(\bot)\}$;
5 while $\exists$ feasible elem $\in L$, incomparable with Frontier do
6     $Frontier \leftarrow Frontier \cup \{maximise(elem)\}$;
7 end
8 return Frontier;
```

Algorithm 2: Maximisation algorithm

```
Input: Feasible element: elem
Result: Maximal feasible element
1 while $\exists$ feasible successor fs of elem do
2     pick element middle such that $fs \sqsubseteq_L middle \sqsubseteq_L \top$;
3     if middle is feasible then
4         $elem \leftarrow middle$;
5     else
6         $elem \leftarrow fs$;
7     end
8 end
9 return elem;
```

elem has any feasible parents (line 1); if this is not the case, *elem* 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 *elem* 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.

Lemma 14 (Correctness of exploration algorithm) When applied to a finite abstraction lattice, Algorithm 1 terminates and returns the set of maximal feasible elements.

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.

C. Selection of Maximal Feasible Elements

Given the 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 a function $cost : L \to \mathbb{N}$ 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. 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 ($cost : T \to \mathbb{N}$), and to define the cost of a lattice element $A \in L$ as $cost(A) = \sum_{t \in A} cost(t)$.

Our abstraction lattice in Fig. 3 has two maximal feasible elements, $\{i_1\}$ and $\{x_1 - i_1, j\}$, that 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 [7], and implemented with the help of "term abstraction").

VII. Integration into a Software Model Checker

A. General Integration

Interpolation abstraction can be applied whenever interpolation is used by a model checker to eliminate spurious counterexamples. To this end, it is necessary to select one or multiple *abstraction points* in the constructed interpolation problem (which might concern an inductive sequence of interpolants, tree interpolants, etc.), and then to define an abstraction lattice for each abstraction point. For instance, when computing an inductive sequence $I_0, I_1, \ldots, I_{10}$ for the conjunction $P_1 \wedge \cdots \wedge P_{10}$, we might select interpolants I_3 and I_5 as abstraction points, choose a pair of abstraction lattices, and add abstraction relations to the conjuncts P_3, P_4, P_5, P_6.

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 can be determined, and subsequently be used to compute abstract interpolants.

B. Abstraction in Eldarica

We have integrated our technique into the predicate abstraction-based model checker Eldarica [2], which uses Horn clauses to represent different kinds of verification problems [3], and solves recursion-free Horn constraints to synthesise new predicates for abstraction [4]. As abstraction points, recurrent control locations in counterexamples are chosen (corresponding to recurrent relation symbols of Horn clauses), which represent loops in a program. Abstraction lattices are powerset lattices over the template terms

$$\begin{aligned} &\{z \mid z \text{ a variable in the program}\} \\ \cup\, &\{x + y,\ x - y \mid x, y \text{ variables assigned in the loop body}\} \end{aligned}$$

In Table I we evaluate the performance of our approach compared to Eldarica without interpolation abstraction, the acceleration-based tool Flata [2], and the Horn engine of Z3 [22] (v4.3.2). Benchmarks are taken from [15], and from a recent collection of Horn problems in SMT-LIB format.[1] They tend to be small (10 – 750 Horn clauses each), but challenging

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

Benchmark	Eldarica		Eldarica-ABS		Flata	Z3
	N	sec	N	sec	sec	sec
C programs from [15]						
boustrophedon (C)	*	*	10	10.7	*	0.1
boustrophedon_expansed (C)	*	*	11	7.7	*	0.1
halbwachs (C)	*	*	53	2.4	*	0.1
gopan (C)	17	22.2	62	57.0	0.4	349.5
rate_limiter (C)	11	2.7	11	19.1	1.0	0.1
anubhav (C)	1	1.7	1	1.6	0.9	*
cousot (C)	*	*	3	7.7	0.7	*
bubblesort (E)	1	2.8	1	2.3	83.2	0.3
insdel (C)	1	0.9	1	0.9	0.7	0.0
insertsort (E)	1	1.8	1	1.7	1.3	0.1
listcounter (C)	*	*	8	2.0	0.2	*
listcounter (E)	1	0.9	1	0.9	0.2	0.0
listreversal (C)	1	1.9	1	1.9	4.9	*
mergesort (E)	1	2.9	1	2.6	1.1	0.2
selectionsort (E)	1	2.4	1	2.4	1.2	0.2
rotation_vc.1 (C)	7	2.0	7	0.3	1.9	0.2
rotation_vc.2 (C)	8	2.7	8	0.2	2.2	0.3
rotation_vc.3 (C)	0	2.3	0	0.2	2.3	0.0
rotation.1 (E)	3	1.8	3	1.8	0.5	0.1
split_vc.1 (C)	18	3.9	17	3.2	*	1.1
split_vc.2 (C)	*	*	18	1.1	*	0.2
split_vc.3 (C)	0	2.8	0	1.5	*	0.0
Recursive Horn SMT-LIB Benchmarks						
addition (C)	1	0.7	1	0.8	0.4	0.0
bfprt (C)	*	*	5	8.3	-	0.0
binarysearch (C)	1	0.9	1	0.9	-	0.0
buildheap (C)	*	*	*	*	-	*
countZero (C)	2	2.0	2	2.0	-	0.0
disjunctive (C)	10	2.4	5	5.0	0.2	0.3
floodfill (C)	*	*	*	*	41.2	0.1
gcd (C)	4	1.2	4	2.0	-	*
identity (C)	2	1.1	2	2.1	-	0.1
mccarthy91 (C)	4	1.4	3	2.4	0.2	0.0
mccarthy92 (C)	38	5.6	7	8.7	0.1	0.1
merge-leq (C)	3	1.1	7	7.0	15.7	0.1
merge (C)	3	1.1	4	4.5	14.7	0.1
mult (C)	*	*	15	52.8	-	*
palidrome (C)	4	1.4	2	2.1	-	0.1
parity (C)	4	1.6	4	2.9	0.8	*
remainder (C)	2	1.1	3	1.6	-	*
running (C)	2	0.9	2	1.7	0.2	0.1
triple (C)	4	2.0	4	5.1	-	0.1

TABLE I
Comparison of Eldarica without interpolation abstraction, Eldarica with **ABS**traction, Flata, and Z3. The letter after the model name distinguishes **C**orrect benchmarks from benchmarks with a reachable **E**rror state. For Eldarica, we give the number of required CEGAR iterations (N), and the runtime in seconds; for Flata and Z3, the runtime is given. Items with "*" indicate a timeout (set to 10 minutes), while - indicates inability to run the benchmark due to lack of support for some operators in the problems. Experiments were done on an Intel Core i7 Duo 2.9 GHz with 8GB of RAM.

for model checkers. We focused on benchmarks on which Eldarica without interpolation abstraction diverges; since interpolation abstraction gives no advantages when constructing long counterexamples, we mainly used correct benchmarks (programs not containing errors). Lattice sizes in interpolation abstraction are typically 2^{15} – 2^{300}; we used a timeout of 1s for exploring abstraction lattices.

The results demonstrate the feasibility of our technique and its ability to avoid divergence, in particular on problems from [15]. Overall, interpolation abstraction only incurs a reasonable runtime overhead. The biggest (relative) overhead could be observed for the rate_limiter example, where some of the feasibility checks for abstraction take long time. Flata is able to handle a number of the benchmarks on which Eldarica times out, but can overall solve fewer problems than Eldarica. Z3 is able to solve many of the benchmarks very quickly, but overall times out on a larger number of benchmarks than Eldarica with interpolation abstraction.

VIII. Conclusion

We have presented a semantic and 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. Various directions of future work are planned: (i) develop further forms of interpolation abstraction, in particular quantified and parametric ones; (ii) application of the framework to programs with arrays and heap; (iii) clearly, our approach is related to the theory of abstract interpretation; we plan whether methods from abstract interpretation can be carried over to our method.

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. This work was supported by the EU FP7 STREP CERTAINTY and by the Swedish Research Council.

References

[1] W. Craig, "Linear reasoning. A new form of the Herbrand-Gentzen theorem," *The Journal of Symbolic Logic*, vol. 22, no. 3, 1957.
[2] H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer, "A verification toolkit for numerical transition systems - tool paper," in *FM*, 2012, pp. 247–251.
[3] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *PLDI*, 2012.
[4] P. Rümmer, H. Hojjat, and V. Kuncak, "Disjunctive interpolants for Horn-clause verification," in *Computer Aided Verification (CAV)*, ser. LNCS, vol. 8044. Springer, 2013, pp. 347–363.
[5] R. Jhala and K. L. McMillan, "A practical and complete approach to predicate refinement," in *TACAS*, 2006, pp. 459–473.
[6] K. L. McMillan, "Quantified invariant generation using an interpolating saturation prover," in *TACAS*, 2008, pp. 413–427.
[7] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina, "Lazy abstraction with interpolants for arrays," in *LPAR*, 2012.
[8] N. Totla and T. Wies, "Complete instantiation-based interpolation," in *POPL*, R. Giacobazzi and R. Cousot, Eds. ACM, 2013, pp. 537–548.
[9] V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher, "Interpolant strength," in *VMCAI*, 2010, pp. 129–145.
[10] S. F. Rollini, O. Sery, and N. Sharygina, "Leveraging interpolant strength in model checking," in *CAV*, 2012, pp. 193–209.
[11] S. Rollini, R. Bruttomesso, and N. Sharygina, "An efficient and flexible approach to resolution proof reduction," in *HVC*, 2010, pp. 182–196.
[12] K. Hoder, L. Kovács, and A. Voronkov, "Playing in the grey area of proofs," in *POPL*, 2012, pp. 259–272.
[13] N. Caniart, E. Fleury, J. Leroux, and M. Zeitoun, "Accelerating interpolation-based model-checking," in *TACAS*, 2008, pp. 428–442.
[14] M. N. Seghir, "A lightweight approach for loop summarization," in *ATVA*, 2011, pp. 351–365.
[15] H. Hojjat, R. Iosif, F. Konecný, V. Kuncak, and P. Rümmer, "Accelerating interpolants," in *ATVA*, 2012, pp. 187–202.
[16] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *FME*, 2001, pp. 500–517.
[17] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko, "Invariant synthesis for combined theories," in *VMCAI*. Springer, 2007.
[18] S. Srivastava and S. Gulwani, "Program verification using templates over predicate abstraction," in *PLDI*, 2009, pp. 223–234.
[19] J. Harrison, *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.
[20] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *CAV*, 1997, pp. 72–83.
[21] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan, "Abstractions from proofs," in *31st POPL*, 2004.
[22] K. Hoder and N. Bjørner, "Generalized property directed reachability," in *SAT*, 2012, pp. 157–171.
[23] A. Brillout, D. Kroening, P. Rümmer, and T. Wahl, "Beyond quantifier-free interpolation in extensions of Presburger arithmetic," in *VMCAI*, ser. LNCS. Springer, 2011.