An Interpolating Decision Procedure for Transitive Relations with Uninterpreted Functions*

Daniel Kroening[1] and Georg Weissenbacher (白傑岳)[1,2,**]

[1] Computing Laboratory, Oxford University
[2] Computer Systems Institute, ETH Zurich

Abstract. We present a proof-generating decision procedure for the quantifier-free fragment of first-order logic with the relations $=$, $\neq$, $\geq$, and $>$ and argue that this logic, augmented with a set of theory-specific rewriting rules, is adequate for bit-level accurate verification. We describe our decision procedure from an algorithmic point of view and explain how it is possible to efficiently generate Craig interpolants for this logic.

Furthermore, we discuss the relevance of the logical fragment in software model checking and provide a preliminary evaluation of its applicability using an interpolation-based program analyser.

1 Introduction

Interpolants play an ever more important role in software and hardware verification [1]. Since interpolants are typically constructed from proofs of inconsistency, interpolation-based verification techniques depend on efficient, proof-generating decision procedures. Interpolating decision procedures have been available for over a decade [2,3], but the field is still advancing rapidly.

McMillan's landmark paper [3] gives an axiomatic description of his interpolating theorem prover FOCI. Recently, more algorithmic descriptions of similar interpolating decision procedures have been published [4,5], indicating that publications that make the material in [3] more accessible are well appreciated.

We present a graph-based interpolating decision procedure for a subset of quantifier free first-order logic with a fixed set of relations, an extension of the logic covered by [4]. We support equality ($=$), disequality ($\neq$), and strong and weak inequality ($>$ and $\geq$, respectively). Furthermore, we provide limited support for interpreted functions such as bit-vector operations. Our presentation emphasises the algorithmic point of view.

Our work is motivated by the discrepancy between the bit-vector interpretation underlying most programming languages and the domains $\mathbb{R}$ or $\mathbb{Z}$ used by many interpolating decision procedures. The decision procedure we present

* Supported by the Semiconductor Research Corporation (SRC) under contract no. 2006-TJ-1539 and by the EU FP7 STREP MOGENTES (project ID ICT-216679).
** Supported by Microsoft Research's European PhD Scholarship Programme.

K. Namjoshi, A. Zeller, and A. Ziv (Eds.): HVC 2009, LNCS 6405, pp. 150–168, 2011.

is *sound* for bit-level formulæ, i.e., if a formula is satisfiable in the bit-vector interpretation, then our algorithm will not conclude that it is unsatisfiable.

Our contribution is a self-contained, algorithmic description of a bit-level accurate decision procedure integrating rewriting rules for theory-specific axioms. We provide a preliminary evaluation of the suitability of our first-order logic fragment for software verification using a re-implementation of the model checking algorithm in [6].

2 Preliminaries

In the following, we define $\mathcal{L}$, a quantifier-free, conjunctive fragment of first-order logic. We restrict the predicates of this language to equality ($=$), disequality ($\neq$), and strong and weak inequality ($>$ and $\geq$, respectively).

Syntax. We fix an enumerable set of variables, function symbols, and constant symbols. Well-formed elements of $\mathcal{L}$ are generated by the following set of rules:

- A *term* t is a constant, a variable, or an application $f(t_1, \ldots t_n)$ of an n-ary function symbol f to terms $t_1, \ldots, t_n$.
- An *atom* $t_1 \rhd t_2$ is a binary relation $\rhd \in \{=, \geq, >, \neq\}$ applied to two terms t_1 and t_2. We do not allow any predicates other than the relations listed above.
- A *formula* F is a conjunction of atoms.

Note that the set of atoms is closed under negation, i.e., the negation $\neg(t_1 \rhd t_2)$ of an atom can be expressed in terms of an atom. Conjunction ($\wedge$) is the only logical connective we allow in $\mathcal{L}$. This is a common restriction for (interpolating) decision procedures for specialised theories, since arbitrary propositional connectives can be handled using the orthogonal approach presented in [3,7].

Interpretations. We use the standard interpretation of the relation symbols $=$ and $\neq$. The relation $\geq$ is a partial order over the (interpreted) domain $\mathcal{D}$, and $t_i > t_j$ denotes $(t_i \geq t_j) \wedge (t_i \neq t_j)$. An *interpreted* n-ary function symbol f has a well-defined function $f^{\mathcal{M}} : \mathcal{D}^n \to \mathcal{D}$ associated to it, whereas an *uninterpreted* function symbol has no other property than its name and arity.

We use $F \models G$ to state that the formula F entails G.

Craig interpolation. Since $\mathcal{L}$ is a fragment of first-order logic, there exists a *Craig interpolant* (a first-order logic formula) for every inconsistent pair of $\mathcal{L}$-formulæ F and G:

Definition 1 (Craig interpolant for $\mathcal{L}$). *Given an unsatisfiable $\mathcal{L}$-formula $F \wedge G$, a* Craig *interpolant is a first-order logic formula I such that*

1. $F \models I$,
2. $G \models \neg I$, *and*
3. *the variables and function symbols I refers to are common to F and G.*

(a) Congruence edge (b) Derived edge

Fig. 1. Congruence edges and derived edges

Remark 1. Note that I is not necessarily a $\mathcal{L}$-formula (and may not even be expressible in $\mathcal{L}$). An example is the pair of formulæ $f(x_0) \neq f(x_5) \wedge x_0 = x_1 \wedge x_2 = x_3 \wedge x_4 = x_5$ and $x_1 = x_2 \wedge x_3 = x_4$ and their interpolant $x_1 \neq x_2 \vee x_3 \neq x_4$.

Graph representation of $\mathcal{L}$ formulæ. The fact that an $\mathcal{L}$-formula F is a conjunction of atoms of the form $t_i \rhd t_j$ enables us to represent F using a graph [8].

Definition 2 ($\mathcal{L}$-graph). *Given a formula F, let $\mathcal{G}_F(V, E)$ be a directed graph, where each term t_i in F corresponds to a node v_i in V, and each atom $t_i \rhd t_j$ corresponds to a $\rhd$-labelled edge $(v_i \xrightarrow{\rhd} v_j) \in E$, $\rhd \in \{=, \geq, >, \neq\}$. Atoms $t_i \rhd t_j$ with a* symmetric *relation $\rhd \in \{=, \neq\}$ additionally contribute an edge $(v_j \xrightarrow{\rhd} v_i)$. For convenience, we use undirected edges to depict equalities and disequalities. In accordance to [3], we write $v_i \simeq v_j$ if and only if $i = j$.*

Due to the presence of functions in $\mathcal{L}$-terms, the congruence relation may give rise to additional equality edges in the graph: The congruence relation satisfies, in addition to the properties of the equality relation, the monotonicity axioms, i.e., for all n-ary functions f, it holds that $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ whenever $s_i = t_i$ holds for all i in $\{1, .., n\}$. We use *congruence edges* to depict such equalities (see Fig. 1a). The dashed arrows indicate that $f(x) = f(y)$ is derived from the equality of the sub-terms $x = y$.

Definition 3 (Contradictory and equality-entailing cycles). *A* contradictory cycle *[8] in an $\mathcal{L}$-graph is a cyclic path consisting of either*

a) edges labelled with $=$ and a single edge labelled with $\neq$, or
b) edges labelled with either $=$ or $\geq$ and at least *one edge labelled with $>$.*

An equality-entailing cycle *in an $\mathcal{L}$-graph is a cyclic path consisting of edges labelled with either $=$ or $\geq$. For any two terms t_i and t_j corresponding to nodes in an equality-entailing cycle, it holds that $t_i \geq t_j$ and $t_j \geq t_i$, and thus $t_i = t_j$.*

We depict derived edges using a graphical representation similar to congruence edges (see Fig. 1b). In this example, the equality $x = y$ is derived from the equality-entailing cycle $x \xrightarrow{\geq} y \xrightarrow{\geq} v \xrightarrow{\geq} u \xrightarrow{\geq} x$.

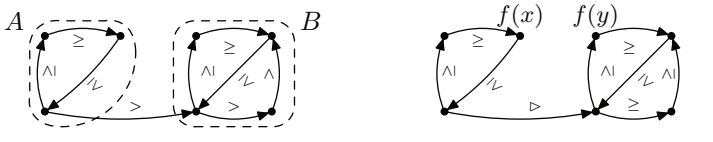
3 A Graph-Based Decision Procedure

We begin this section with a brief outline of our decision procedure for $\mathcal{L}$-formulæ followed by a detailed description of the proof-generating algorithm. Let $\mathcal{G}(V, E)$ be the $\mathcal{L}$-graph for a given formula F. The decision procedure is subdivided into two phases:

1. In the first phase, the algorithm searches for contradictory or equality-entailing cycles with edges labelled $=$, $\geq$, and $>$ (Def. 3*a*) in the graph $\mathcal{G}(V, E^{\geq})$, where $E^{\geq}$ denotes $E \setminus \{(v_i \xrightarrow{\neq} v_j) \in E\}$. If a contradictory cycle exists, the algorithm terminates. Otherwise, the procedure adds to E the edges $v_i \xrightarrow{=} v_j$ and $v_j \xrightarrow{=} v_i$ for all nodes v_i, v_j adjacent in an equality-entailing cycle.
2. In the second phase, additional equalities are inferred by means of constant propagation and congruence closure and searches for contradictory cycles with edges labelled $=$ or $>$ (Def. 3*b*) in the graph $\mathcal{G}(V, E^{\neq})$, where $E^{\neq} = \{(v_i \rhd v_j) \in E \mid \rhd \in \{=, \neq\}\}$.

The phases are iterated until no new equalities can be inferred. Both phases use well-known and efficient graph algorithms such as Tarjan's algorithm for the computation of *strongly connected components* (SCCs) and a graph-based *union-find* data structure. In a pre-processing step, we form two (possibly non-disjoint) sets of the atoms in F, one of which contains the inequalities and equalities, and one which contains equalities and disequalities.

Phase I: Inequalities. Let $\mathcal{G}(V, E^{\geq})$ be the $\mathcal{L}$-graph corresponding to the equality and inequality atoms of F. Using Tarjan's algorithm, we compute all strongly connected components in $\mathcal{G}(V, E^{\geq})$ and classify them as contradictory or equality-entailing cycles, respectively:



(a) SCCs in $\mathcal{L}$-graphs (b) Congruences and SCCs

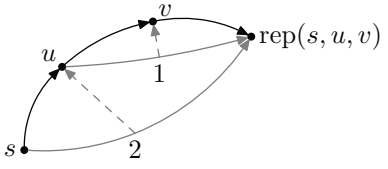
Fig. 2. Strongly connected components (SCCs) in $\mathcal{L}$-graphs

1. A SCC is contradictory if it contains at least one edge $v_i \xrightarrow{>} v_j$ (see component B in Fig. 2a). Then, any path from v_j to v_i forms a contradictory cycle with $v_i \xrightarrow{\geq} v_j$. If our algorithm finds a contradictory SCC, we compute the shortest such path and report it as a proof of inconsistency.

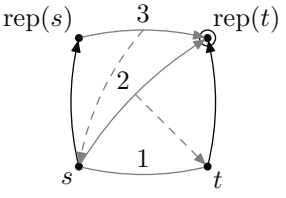
2. A SCC is equality-entailing if it contains no edge labelled with $>$ (see component A in Fig. 2a or the SCCs in Fig. 2b). In this case, we conclude that for any edge $v_i \xrightarrow{\triangleright} v_j$ in the SCC $t_i = t_j$ holds for the corresponding terms. The derived equalities are passed on to the second phase.

Phase II: Equalities and Disequalities. The second phase starts with computing the equivalence closure of the equality atoms (and the equalities derived in the first phase). For this purpose, we use a *proof-generating* union-find data structure that incrementally constructs an $\mathcal{L}$-graph $\mathcal{G}(V, E^{=})$, where $E^{=}$ denotes a set of edges labelled with $=$. In the following, we present the modifications necessary to generate a proof of inconsistency. In a union-find data structure, each equivalence class corresponds to a sub-graph of $\mathcal{G}(V, E^{=})$ identified by its *representative*, and each node which is not a representative holds a reference to its parent node (indicated by an directed edge in our illustrations). The data structure supports two operations:

1. *Find*(v_i) returns the representative of the node v_i.
2. *Union*(v_i, v_j) adds an (undirected) equality edge to the graph $\mathcal{G}(V, E^{=})$ and merges the two equivalence classes containing v_i and v_j, respectively.



(a) Path compression



(b) Triangulation

assert $v_1 \not\simeq v_2 \wedge r_1 \not\simeq r_2$
if $v_1 \simeq r_1$ **then**
  if $v_2 \not\simeq r_2$ **then**
    $E^{=} := E^{=} \cup \{r_2 \xrightarrow{\langle v_2 \rangle} r_1\}$
  end if
else
  if $v_2 \not\simeq r_1$ **then**
    $E^{=} := E^{=} \cup \{v_2 \xrightarrow{\langle v_1 \rangle} r_1\}$
  end if
  if $v_2 \not\simeq r_2$ **then**
    $E^{=} := E^{=} \cup \{r_2 \xrightarrow{\langle v_2 \rangle} r_1\}$
  end if
end if

(c) Implementation of 3b

Fig. 3. An illustration of union-find operations

The *Find*(v_i) operation performs *path compression* in order to reduce the computational effort in case of repeated queries for v_i. During this process, it adds new derived edges to $E^{=}$, which connect v_i directly with its representative. This is illustrated by the example in Fig. 3a. *Find* follows the parent nodes until it reaches the representative. In Fig. 3a, the call to *Find*(v_1) results in two recursive calls *Find*(v_2) and *Find*(v_3). The latter call returns v_4 as the representative for v_3. We add $v_2 \xrightarrow{\langle v_3 \rangle} v_4$ to $E^{=}$ (step 1 in Fig. 3a) and replace the parent v_3 with v_4. Here, the label $\langle v_3 \rangle$ is used to memorise the fact that $v_2 \xrightarrow{=} v_3$

derives from $v_2 \xrightarrow{=} v_3 \xrightarrow{=} v_4$ (visualised by the dashed arrow). Finally, $\mathit{Find}(v_2)$ yields v_4 and we add $v_1 \xrightarrow{\langle v_3 \rangle} v_4$ to $E^{=}$ and replace the parent v_2 with v_4. Thus, $\mathit{Find}(v_1)$ returns v_4.

The $\mathit{Union}(v_1, v_2)$ operation merges two equivalence classes with the representatives r_1, r_2 (obtained using *Find*). We assume that redundant unions are ignored, i.e., $v_1 \not\simeq v_2$ and $r_1 \not\simeq r_2$. Consider the example in Fig. 3b. We add the edge $v_1 \xrightarrow{=} v_2$ (step 1) and conclude that the terms corresponding to r_1 are r_2 equivalent. The algorithm chooses a new representative (r_1 in our example), favouring nodes with a higher in-degree. The resulting edge $v_2 \xrightarrow{\langle v_1 \rangle} r_1$ is labelled accordingly in step 2, in order to memorise its derivation. Finally, we connect r_2 and r_1; the corresponding edge derives from $r_2 \xrightarrow{=} v_2 \xrightarrow{=} r_1$.

Observe that *Union* triangulates the sub-graph spanning $V = \{v_1, v_2, r_1, r_2\}$. Fig. 3c shows the general algorithm for this triangulation (where r_1 is the representative node with the higher in-degree), which is a constant time operation.

Using *Union*, we compute the equivalence closure for F by adding all equivalence atoms and derived equalities to $\mathcal{G}(V, E^{=})$. We can now efficiently query whether a disequality $t_i \neq t_j$ contradicts the equality relations stored in $\mathcal{G}(V, E^{=})$ by checking whether $\mathit{Find}(v_i) \simeq \mathit{Find}(v_j)$. If this is the case, we obtain a contradictory cycle $v_i \xrightarrow{\neq} v_j \xrightarrow{=} r \xrightarrow{=} v_i$. From this cycle, we obtain a proof for the inconsistency by repeatedly expanding derived edges $v_i \xrightarrow{\langle v_j \rangle} v_k$ to $v_i \xrightarrow{=} v_j \xrightarrow{=} v_k$. Edges derived in Phase I are justified by their respective equality-entailing cycles.

Congruence closure. The decision procedure described above lacks a provision for deriving congruence edges (Fig. 1a) and is therefore not sufficient to support uninterpreted functions. An equality relation $t_i = t_j$ in the congruence graph $\mathcal{G}(V, E^{=})$ gives rise to a congruence edge representing $f(t_i) = f(t_j)$, which, in return, may entail additional equality relations in $\mathcal{G}(V, E^{=})$. Therefore, we use an incremental *congruence closure* algorithm (following the ideas presented in [9]) that is closely intertwined with the construction of the $\mathcal{L}$-graph for equalities.

The algorithm uses the union-find data-structure representing $\mathcal{G}(V, E^{=})$. It *indexes* each representative in $\mathcal{G}(V, E^{=})$ with a term t_c. Thus, all terms in an equivalence class of $\mathcal{G}(V, E^{=})$ are associated with the same term t_c. If the equivalence class contains an interpreted constant (e.g., a numeral), we choose it as the index term,[1] otherwise, we use the term corresponding to the representative of the equivalence class as index. In this setting, an equivalence class containing terms with function symbols represents a set of congruence relations. Consider two terms $f(t_i)$ and $f(t_j)$, where t_i and t_j have the same representative indexed with t_c. Then $f(t_i)$ and $f(t_j)$ belong to the same equivalence class.

In addition, we maintain a function $\mathit{Lookup}(f(t_c))$ which maps $f(t_c)$ to a term $f(t_i)$ such that t_i belongs to an equivalence class indexed with t_c, or $\perp$ if there is no such term in $\mathcal{G}(V, E^{=})$.

[1] Note this constant is unique, since an equivalence class that contains two constants with a different interpretation contains a contradictory cycle.

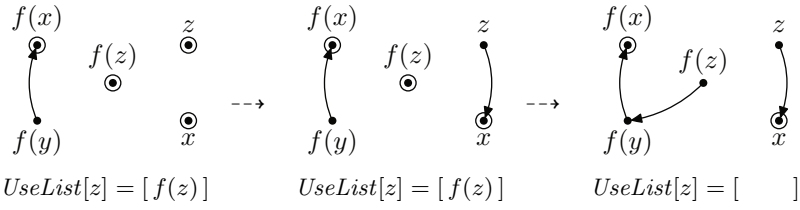


Fig. 4. A 3-step example illustrating the congruence closure algorithm

The *Union* operation potentially changes the representatives of the equivalence classes in $\mathcal{G}(V, E^{=})$. Therefore, the algorithm maintains for each index term t_c a list *UseList*(t_c) of terms that contain a sub-term indexed with t_c. This list is updated whenever *Union* merges two equivalence classes. W.l.o.g., assume that *Union* merges an equivalence class indexed with t_c with an equivalence class indexed with t'_c, choosing the latter term as the new index. Then, for each $f(t_i) \in \mathit{UseList}(t_c)$, where t_c is the index term associated with t_i, the algorithm proceeds as follows:

- If *Lookup*($f(t'_c)$) returns $f(t_j)$, it uses *Union* to add a the congruence edge for $f(t_i) = f(t_j)$ to $\mathcal{G}(V, E^{=})$ and memorises that the edge is derived from $t_i = t_j$. Furthermore, $f(t_i)$ is moved from *UseList*(t_c) to *UseList*(t'_c).
- If *Lookup*($f(t'_c)$) returns $\perp$, it sets *Lookup*($f(t'_c)$) to $f(t_i)$ and moves $f(t_i)$ to *UseList*(t'_c).

Example 1. Consider a union-find data structure with four equivalence classes $\{f(x), f(y)\}$, $\{f(z)\}$, $\{x\}$, and $\{z\}$ (see Fig. 4, on the left). *UseList*$[z]$ contains $f(z)$, since z is a sub-term of $f(z)$. Adding $x = z$ yields a new equivalence class $\{x\} \cup \{z\}$. Assume that the representative of the resulting equivalence class $\{x, z\}$ is x and that *Lookup*$(f(x)) = f(y)$. Then the algorithm infers $f(z) = f(y)$. $\lhd$

The extension to n-ary functions is straight-forward. An efficient implementation based on *currifying* is presented in [9].

Bit-vector theory axioms, constant propagation, and interpreted functions. Our decision procedure provides limited support for the theory of bit-vectors by integrating a small set of bit-vector axioms and rewriting rules. Furthermore, whenever possible, it uses interpreted functions and constants in order to simplify terms. This is achieved by the following mechanisms:

1. We order all interpreted constants $c_1, \ldots, c_n$ processed in Phase I and add $n-1$ inequality relations of the form $c_i < c_{i+1}$, $1 \leq i < n$ to $\mathcal{G}(V, E^{\geq})$ *before* computing the SCCs.
2. In Phase II, if *Union* is applied to two terms indexed with different interpreted constants c_1 and c_2, we introduce the disequality $c_1 \neq c_2$.
3. Let T be the set of terms corresponding to the nodes in $\mathcal{G}(V, E^{=})$. For each $f(t_i) \in T$ such that f is an interpreted function symbol in a given theory $\mathcal{T}$

$$(t_2 + c) \neq t_2 \text{ if } c \neq 0 \bmod 2^m \qquad (t \ll c) = (t + t) \text{ if } c = 1$$
$$(t_2 + c) = t_2 \text{ if } c = 0 \bmod 2^m \qquad (t \ll c) = (2^c \cdot t) \text{ if } 1 < c < m$$

Fig. 5. Two examples for rewriting rules for m-bit variables

$$\frac{t_1 = t_2 \,\&\, t_3}{t_1 \leq t_2 \quad t_1 \leq t_3} \qquad \frac{t_1 = t_2 \,|\, t_3}{t_1 \geq t_2 \quad t_1 \geq t_3} \qquad \frac{t_1 + t_2 = t_1}{t_2 = 0}$$

Fig. 6. Examples of axioms for bit-vector operations

and t_i is a term indexed with an interpreted constant c, we check whether $f(c)$ can be simplified to a term t_j not containing any variables or function symbols that do not occur in $f(c)$. If this is the case, and $t_j \in T$ or t_j is an interpreted constant, we add the equivalence relations $f(t_i) = f(c)$ (derived from $t_i = c$) and $f(c) = t_j$ (a tautology in $\mathcal{T}$) to $\mathcal{G}(V, E^{=})$. This technique allows us to perform *bit-level-accurate* simplifications of terms.

4. We apply a fixed set of rewriting rules of the form $t \rhd t'$ to all terms t, where t' is the term obtained by applying the rule to t. All rules have the property that they do not introduce variables. Examples of such rules are listed in Fig. 5. If t and t' correspond to nodes in $\mathcal{G}(V, E^{=})$, we add the relation $t \rhd t'$.
5. Axioms of the form $(t_1 \rhd_1 t_2) \vdash (t_3 \rhd_2 t_4)$ may be applied if t_3 and t_4 refer to a subset of the non-logical symbols in t_1 and t_2. Examples of such axioms are provided in Fig. 6.[2]

Combining both phases. As explained above, equality relations derived from equality-entailing cycles in Phase I are passed on to Phase II. Now consider the $\mathcal{L}$-graph in Fig. 2b. Adding the congruence edge corresponding to $f(x) = f(y)$ results in a new SCC, which, depending on the label $\rhd$ in Fig. 2b, is either contradictory or equality-entailing. Therefore, the congruence edges generated in Phase II must be added to $\mathcal{G}(V, E^{\geq})$, necessitating an additional iteration of Phase I. The two phases need to be iterated until no more new congruence edges are generated. Since both phases are exchanging equalities exclusively, our implementation is essentially a Nelson-Oppen-style decision procedure.

Complexity. Tarjan's algorithm applied in Phase I has a run-time linear in the number n of edges of the graph. The computation of the equivalence closure in the second phase takes $O(n \cdot \alpha(n))$ time, where α is the inverse of the Ackermann function $A(n, n)$. The congruence closure is of complexity $O(n \cdot \log n)$ [9]. Thus, a single iteration of Phase I and Phase II takes $O(n \cdot \log n)$ time.

It remains to determine how often the phases need to be iterated. Since the algorithm never adds redundant congruence edges, the congruence closure adds at most $O(n)$ equalities (see [9]). Due to the restrictions on the application of

[2] The naïve application of such axioms increases the complexity of the algorithm significantly. Therefore, we apply each axiom only once in an initial rewriting phase.

rules and axioms, rewriting interpreted functions increases the number of subterms by at most a constant factor. Altogether, we face a run-time complexity of $O(n^2 \cdot \log n)$ for our decision procedure.

Finally, the extraction of an explanation from a contradictory cycle can be performed in $O(n \cdot \log n)$ time, since the derived edges form a tree.

Proofs of inconsistency. We review the artefacts generated by our decision procedure. A proof of inconsistency of an $\mathcal{L}$-formula F is a *contradictory cycle* comprising

- edges directly corresponding to relations in F,
- edges derived from equality-entailing cycles, and
- congruence edges, derived from a number of equality relations.

In the next section we explain how a Craig interpolant can be constructed from such a proof of inconsistency.

4 Extracting Interpolants from Contradictory Cycles

This section introduces the concept of coloured $\mathcal{L}$-graphs and explains how interpolants can be constructed from contradictory cycles in such a coloured graph.

Colouring $\mathcal{L}$-graphs. Given an $\mathcal{L}$-formula $F \wedge G$, we say that a node v_i of the corresponding graph $\mathcal{G}(V, E)$ is F-colourable if the corresponding term t_i refers only to variables and function symbols in F; similarly for G. We use V_F and V_G to refer to the set of F-colourable and G-colourable nodes, respectively. This definition splits $V = V_F \cup V_G$ into two non-disjoint sets of vertices. It leaves us a choice for a subset $V_S \stackrel{def}{=} (V_F \cap V_G)$ of V. We refer to V_S as *shared vertices.*

An edge $v_i \stackrel{\triangleright}{\rightarrow} v_j$ is F-colourable if and only if $\{v_i, v_j\} \subseteq V_F$; analogously for G. We use E_F (E_G) to refer to the F-colourable (G-colourable, respectively) edges in E. An edge is colourable if it is either F-colourable or G-colourable. The edges of the initial $\mathcal{L}$-graph $\mathcal{G}(V, E)$, in which each edge corresponds to an atom in $F \wedge G$, are always colourable. This is not necessarily the case for the graph that we obtain by computing the congruence closure (in Phase II). Consider the nodes labelled $f(x)$ and $f(y)$ in the $\mathcal{L}$-graph in Fig. 2b. Assume that the variable x occurs only in F and y occurs only in G. If we deduce $f(x) = f(y)$ from $x = y$, then the corresponding edge is not colourable.

It is, however, possible to transform a congruence-closed $\mathcal{L}$-graph into a colourable graph [4,10]. We provide a constructive proof based on structural induction over an $\mathcal{L}$-graph with congruence edges:

1. *Base case.* Colour the equality edges of the $\mathcal{L}$-graph according to their respective atoms in the formula $F \wedge G$.
2. *Induction step.* The argument is split into two cases:
   (a) *Derived edges.* For each edge $v_i \stackrel{=}{\rightarrow} v_j$ derived from an equality-entailing cycle, there exists an edge $v_i \stackrel{\triangleright}{\rightarrow} v_j$ ($\triangleright \in \{\geq, =\}$) in that cycle, which is, by the induction hypothesis, colourable. Let $v_i \stackrel{=}{\rightarrow} v_j$ take the colour of that edge.

Table 1. Rules for labelling contracted edges

	$=$	$\neq$	$\geq$	$>$
$=$	$=$	$\neq$	$\geq$	$>$
$\geq$	$\geq$	$\bot$	$\geq$	$>$
$>$	$>$	$\bot$	$>$	$>$

In order to label facts, the labels of the edges on a path are merged according to the rules to the left. By construction, the decision procedure described in Section 3 guarantees that no fact in a proof of inconsistency has an undefined ($\bot$) label.

(b) *Congruence edges.* Pick any non-colourable congruence edge with nodes $v_{f(x)}$ and $v_{f(y)}$ labelled $f(x)$ and $f(y)$, respectively. By the induction hypothesis, all edges in the path $v_x \to \ldots \to v_y$ entailing $x = y$ can be coloured. Since v_x and v_y are of different colour, there is a path prefix $v_x \to \ldots \to v_z$ such that all nodes in the prefix are of the same colour and $v_z \in V_S$. Let z be the term that corresponds to v_z. Then, the term $f(z)$ refers only to non-logical symbols common to F and G. Introduce a new node $v_{f(z)}$ representing $f(z)$ and add an equality edge $v_{f(x)} \to v_{f(z)}$ justified by $v_x \to \ldots \to v_z$, and a new congruence edge $v_{f(z)} \to v_{f(y)}$ justified by $v_z \to \ldots \to v_y$. All these new elements are colourable.

This proof translates into an algorithm of complexity $O(n \cdot \log n)$. The transformation yields a graph representing a formula *equisatisfiable* with $F \wedge G$, i.e., the modified graph contains a contradictory cycle if and only if the original congruence-closed graph $\mathcal{G}(V, E)$ contains one.

It is straight-forward to extend this argument to the edges introduced by the term rewriting rules and axioms in Section 3. Consider, w.l.o.g., an F-coloured node v_i corresponding to a term t, and a node v_j corresponding to the rewritten term t'. Due to the restriction that the rewriting rule $t \rightsquigarrow t'$ must not introduce new non-logical symbols,[3] the edge $v_i \to v_j$ can be coloured with 'F'. A similar argument holds for axioms, which do not change the colour of the affected edge.

This line of reasoning leads to the following observation:

Lemma 1. *A proof of inconsistency, which is a sub-graph of the congruence-closed $\mathcal{L}$-graph $\mathcal{G}(V, E)$ obtained using the algorithm in Section 3, can be transformed into a colourable graph.*

Furthermore, given that an $\mathcal{L}$-graph $\mathcal{G}(V, E)$ represents a formula $F \wedge G$, which is a *conjunction* of atoms, the formula represented by a sub-graph is implied by $F \wedge G$. Thus, the proof of inconsistency is implied by the original formula $F \wedge G$.

Interpolants from coloured inconsistency proofs. Given a coloured proof of inconsistency, it is now possible to *factorise* this graph according to the colour of its edges. Accordingly, a *factor* of a path in this graph is a maximal sub-path consisting of edges of equal colour. If we contract a factor $v_1 \overset{\rhd_1}{\to} \ldots \overset{\rhd_{n-1}}{\to} v_n$, we obtain a *fact* $v_1 \overset{\rhd}{\to} v_n$. The label $\rhd$ of this fact is determined by iteratively merging the labels along the path according to the rules in Table 1.

[3] Interpreted function symbols and constants are considered logical symbols.

Facts over the shared vocabulary V_S are the basic building blocks of interpolants for $\mathcal{L}$-graphs. In general, however, it is not possible to represent an interpolant for $F \wedge G$ as an $\mathcal{L}$-graph or as an $\mathcal{L}$-formula (see Remark 1 in Section 2). Intuitively, the reason is that the proof of inconsistency is a result of a mutual interplay[4] of facts derived from F-coloured as well as from G-coloured edges. An F-coloured congruence edge may be derived from a path that contains edges corresponding to atoms in G. This prevents us from extracting a contradictory sub-graph of the proof that is derived exclusively from F.

We account for the interrelation between F-coloured and G-coloured facts by introducing conditions and premises for facts in $\mathcal{L}$-graphs.

Definition 4 (Conditions for facts, edges). *Let $E = E_F \uplus E_G$ and $V = V_F \cup V_G$ be a colouring of the edges and vertices of a proof of inconsistency for $F \wedge G$. A* condition *for a fact (or edge) $v_i \overset{=}{\rightarrow} v_j$ is a (possibly empty) set C of facts obtained from factorised and contracted paths in E such that one of the following conditions holds:*

- *$C = \emptyset$ and $v_i \overset{=}{\rightarrow} v_j$ is a contraction of edges corresponding to atoms in $F \wedge G$.*
- *$v_i \overset{=}{\rightarrow} v_j$ can be derived from the $\mathcal{L}$-graph $\mathcal{G}(V, C)$ by means of equality and congruence closure and equality-entailing cycles.*

We refer to the subset of F-coloured (G-coloured) facts in C as F-condition (G-condition, respectively).

The facts in a proof of inconsistency as constructed by the decision procedure in Section 3 comprise congruence edges, edges derived from equality-entailing cycles, and "basic" edges corresponding to atoms in the original formula $F \wedge G$. The conditions for basic edges and facts comprising only basic edges are defined to be $C = \emptyset$ in Def. 4. For the remaining artefacts, we construct a set of conditions C as follows:

1. *Congruence edges.* For a congruence edge, C is the set of facts obtained by factorising and contracting the path the congruence edge is derived from.
2. *Edges derived from equality-entailing cycles.* For an edge derived from an equality-entailing cycle, C is the set of facts obtained by factorising and contracting that cycle.
3. *Facts.* The condition for a fact $v_1 \rightarrow v_n$ obtained by contracting the path $v_1 \rightarrow \ldots \rightarrow v_n$ is $\bigcup_{i \in \{1..n-1\}} C_i$, where C_i is a condition for $v_i \rightarrow v_{i+1}$.

The correctness of this construction follows immediately from Def. 4 and the definition of congruence edges and derived edges.

A premise denotes a *recursively closed* set of conditions, in which the derived facts are in turn justified by their respective conditions:

[4] This process can also be formalised as a cooperative two-player game [4].

Definition 5 (Premises for facts). *The F-premise for a fact $v_i \xrightarrow{=} v_j$ of colour G is the set F-premise$(v_i \xrightarrow{=} v_j)$ of F-coloured facts defined as*

$$F\text{-premise}\,(v_i \xrightarrow{=} v_j) \stackrel{\text{def}}{=} (F\text{-condition for } v_i \xrightarrow{=} v_j) \cup \bigcup \{F\text{-premise}\,(v_n \to v_m) \mid v_n \to v_m \in (G\text{-condition for } v_i \xrightarrow{=} v_j)\}\,.$$

The definition of the G-premise for F-coloured facts is symmetric.

Premises can be seen as a form of rely-guarantee reasoning. F-premises take the role of ρ in McMillan's interpolations [3], and G-premises correspond to justifications in [4].

Lemma 2. *Let C be the F-condition of a G-coloured fact (or edge) $v_i \xrightarrow{=} v_j$ in a coloured proof of inconsistency $\mathcal{G}(E, V)$, where $E = E_F \uplus E_G$ and $V = V_F \cup V_G$. For all $(v_n \to v_m) \in C$ it holds that $v_n, v_m \in V_S$.*

It follows immediately that F-premises and G-premises refer only to the shared vertices of a proof of inconsistency (cf. Lemma 2(*iii*) in [4]).

Definition 6 ($\mathcal{L}$-graph-based interpolant). *Let $\mathcal{G}(V, E)$ be a proof of inconsistency for $F \wedge G$ and let $E = E_F \uplus E_G$ and $V = V_F \cup V_G$, $V_S = V_F \cap V_G$ be a colouring of its edges and vertices. A* $\mathcal{L}$-graph-based interpolant *is a pair $\langle \mathcal{I}, \mathcal{J} \rangle$ of sets such that the following mutual conditions hold:*

1. *$\mathcal{J}$ is a set of pairs $\langle P, v_i \to v_j \rangle$, and for each $\langle P, v_i \to v_j \rangle \in \mathcal{J}$ it holds that*
   (a) $P \subseteq \mathcal{I}$ is the F-premise for the G-coloured fact $v_i \to v_j$, and
   (b) for all $v_n \to v_m \in \mathcal{I}$, the G-premise for $v_n \to v_m$ is a subset of

$$\{v_k \to v_l \mid \langle P, v_k \to v_l \rangle \in \mathcal{J}\}\,.$$

2. *$\mathcal{I}$ is a set of F-coloured facts obtained by contracting edges in E_F, and the graph*

$$\mathcal{G}\,(V_S, \mathcal{I} \cup \{v_i \to v_j \mid \langle P, v_i \to v_j \rangle \in \mathcal{J}\}) \tag{1}$$

   contains a contradictory cycle.
3. *For all $v_n \xrightarrow{\rhd} v_m$ in $\mathcal{I} \cup \{v_i \xrightarrow{\rhd} v_j \mid \langle P, v_i \xrightarrow{\rhd} v_j \rangle \in \mathcal{J}\}$ it holds that either*
   (a) $v_n, v_m \in V_S$, or
   (b) $v_n \simeq v_m$ and $\rhd \in \{>, \neq\}$.

Fig. 7 shows an algorithm that extracts a pair $\langle \mathcal{I}, \mathcal{J} \rangle$ from a proof of inconsistency. We argue that $\langle \mathcal{I}, \mathcal{J} \rangle$ is an $\mathcal{L}$-graph-based interpolant:

1. Since the factorisation and contraction preserves the structure of the graph, the graph $\mathcal{G}(V_F \cup V_G, E_F \uplus E_G)$ contains a contradictory cycle of facts (possibly degenerate, i.e., $v_i \xrightarrow{\rhd} v_i$, $\rhd \in \{>, \neq\}$). Therefore, E_C exists.

```
 1: let G(V_F ∩ V_G, E_F ∪ E_G) be the factorised and contracted proof
 2: let E_C be the facts in the contradictory cycle of G(V_F ∪ V_G, E_F ∪ E_G)
 3: W := E_C, I := ∅, J := ∅
 4: while (W ≠ ∅) do
 5:     remove v_i → v_j from W
 6:     if v_i → v_j is G-coloured then
 7:         P := F-premise (v_i → v_j)
 8:         J := J ∪ {⟨P, v_i → v_j⟩}
 9:     else
10:         P := G-premise (v_i → v_j)
11:         I := I ∪ {v_i → v_j}
12:     end if
13:     W := W ∪ P
14: end while
```

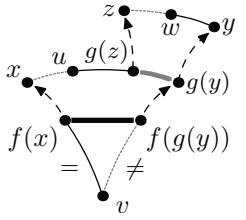
Fig. 7. Computing an $\mathcal{L}$-graph-based interpolant

2. Observe that the algorithm maintains the following invariants:
   (a) For each $\langle P, v_i \to v_j \rangle \in \mathcal{J}$, P is an F-premise of $v_i \to v_j$ and a subset of $\mathcal{W} \cup \mathcal{I}$ (established in line 3 and maintained by lines 7, 8, and 13).
   (b) For each $v_i \to v_j \in \mathcal{I}$, the G-premise of $v_i \to v_j \in \mathcal{I}$ is a subset of $\mathcal{W} \cup \{v_n \to v_m \,|\, \langle P, v_n \to v_m \rangle \in \mathcal{J}\}$. This is established in line 3 and maintained by the statements in lines 10, 11, and 13.
   (c) $\mathcal{G}(V_S, \mathcal{W} \cup \mathcal{I} \cup \{v_i \to v_j \,|\, \langle P, v_i \to v_j \rangle \in \mathcal{J}\})$ contains a contradictory cycle. This invariant is established in line 3.

   Upon termination of the algorithm, $\mathcal{W} = \emptyset$ holds. Together with $\mathcal{W} = \emptyset$, the invariant (2a) implies condition (1a) and the invariant (2b) implies condition (1b) in Def. 6. Furthermore, it follows from the invariant (2c) that condition (2) in Def. 6 is fulfilled.
3. Due to the tree-structured derivations in the proof, the algorithm terminates.
4. The sets $\mathcal{I}$ and $\mathcal{J}$ contain only
   (a) edges $v_i \to v_j$ from the factorised and contracted contradictory cycle E_C of the proof of inconsistency (lines 2 and 3), and
   (b) G-premises (F-premises) for F-coloured (G-coloured) facts.

   According to Lemma 2, all facts in the premises (4b) are edges with endpoints $v_i, v_j \in V_S$. If E_C contains F-coloured as well as G-coloured facts, then the facts (4a) must be edges connecting vertices in V_S. Otherwise, E_C contains a single degenerate edge $v_i \overset{\triangleright}{\to} v_i$, where v_i is not necessarily an element of V_S. Therefore, condition 3 in Def. 6 holds.

The interpolant I for an $\mathcal{L}$-formula $F \wedge G$ may not be expressible in $\mathcal{L}$ (see Remark 1). We can, however, translate the $\mathcal{L}$-graph-based interpolant into an $\mathcal{L}$-formula with disjunctions:



(a) A proof of inconsistency

$$f(g(y)) = v \wedge w = y \vee$$
$$f(g(y)) = v \vee$$
$$(x \neq u) \vee$$
$$(w = y) \wedge g(z) \neq g(y)$$

(b) Interpolant for Fig. 8a

Fig. 8. An example of an interpolant for an inconsistency proof

$$I \stackrel{\text{def}}{=} \underbrace{\bigwedge_{v_i \overset{\triangleright}{\to} v_j \in \mathcal{I}} (t_i \triangleright t_j)}_{(a)} \vee \underbrace{\bigvee_{\langle P, v_n \overset{\triangleright}{\to} v_m \rangle \in \mathcal{J}} \left(\bigwedge_{(v_i \overset{\triangleright_P}{\to} v_j) \in P} (t_i \triangleright_P t_j) \right) \wedge \neg(t_n \triangleright t_m)}_{(b)} \quad (2)$$

We simplify all terms of the form $t_i \triangleright t_i$ to **false** if $\triangleright \in \{>, \neq\}$ and to **true** if $\triangleright \in \{\geq, =\}$.

Example 2. Consider the proof of inconsistency shown in in Fig. 8a. Contracting the inconsistent cycle yields $f(g(y)) \neq v$ (G-coloured) and $f(g(y)) = v$ (F-coloured) *under the condition* that $u = g(z)$ (F-coloured), and $g(z) = g(y)$ (G-coloured) hold. The condition for $g(z) = g(y)$, in turn, is that $z = w$ and $w = y$ holds, where the latter fact is F-coloured. The resulting interpolant is shown in Fig. 8b. ◁

Finally, we claim that I as defined in (2) is indeed an interpolant for $F \wedge G$.

Theorem 1. *Given an $\mathcal{L}$-graph-based interpolant $\langle \mathcal{I}, \mathcal{J} \rangle$ for an $\mathcal{L}$-formula $F \wedge G$, the formula (2) is an interpolant for $F \wedge G$.*

Let us provide an intuitive explanation of Formula (2) before we proceed to the proof of Theorem 1. The formula is split into two sub-formulæ (a) and (b): Condition (1b) in Def. 6 guarantees that ($2a$) holds if

$$\bigwedge_{\langle P, v_n \overset{\triangleright}{\to} v_m \rangle \in \mathcal{J}} (t_n \triangleright t_m) \quad (3)$$

and F (i.e., the F-coloured atoms in $F \wedge G$) hold.

Formula (2b) takes the rôle of the interface in rely-guarantee reasoning and challenges G to contradict one of the atoms in Formula (3). The F-premises of these G-coloured atoms are a subset of $\mathcal{I}$, and therefore implied by ($2a$) due to condition ($1a$) in Def. 6. The G-premises of the facts in $\mathcal{I}$ are in turn implied by Formula (3). The tree-structured derivations of congruence edges and derived edges (generated by algorithm in Fig. 7) prevent circular reasoning. The resulting

tree-structure of these premises is illustrated in Fig. 9: The G-coloured facts $e_1, \ldots, e_5$ derived from the F-premises at the leaves in turn form the G-premise at an inner node of the tree, and so on. We show that this structure prevents G from contradicting Formula (3).

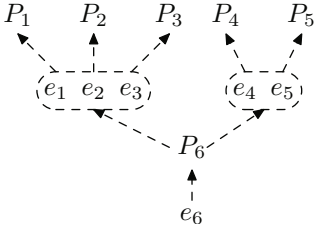


Fig. 9. Tree-structure of the premises in an interpolant

We are now in a position to show the correctness of Theorem 1.

Proof: We review the conditions of Def. 1 in Section 2:

1. $F \models I$. Consider that the G-premises (3) for the F-facts in (2a) hold. W.l.o.g., pick an edge $v_i \xrightarrow{\triangleright} v_j$ from $\mathcal{I}$. Since (3) holds, the G-premise for $v_i \xrightarrow{\triangleright} v_j$ holds. We show that the F-condition (Def. 4) for $v_i \xrightarrow{\triangleright} v_j$ is implied by F and (3) by means of induction.
   - *Base case.* The height of the tree-shaped derivation of $v_i \xrightarrow{\triangleright} v_j$ is one; Thus, the F-condition of $v_i \xrightarrow{\triangleright} v_j$ is a subset of the atoms in F.
   - *Hypothesis.* The F-condition of $v_i \xrightarrow{\triangleright} v_j$ is implied by F and (3) if the height of the tree-shaped derivation is $n-1$ or less.
   - *Induction step.* The height of the derivation of $v_i \xrightarrow{\triangleright} v_j$ is n. W.l.o.g., pick a fact $v_n \rightarrow v_m$ from the F-condition of $v_i \xrightarrow{\triangleright} v_j$. The height of the tree-shaped derivation of this fact is $n-1$ or less. The G-condition for $v_n \rightarrow v_m$ holds because of Def. 5, condition (1b) in Def. 6, and the assumption that (3) holds. The F-condition of $v_n \rightarrow v_m$ holds by our induction hypothesis, and therefore $v_n \rightarrow v_m$ and the F-condition of $v_i \xrightarrow{\triangleright} v_j$ must hold.

   Therefore, F and (3) imply (2a). Otherwise, at least one atom $t_i \triangleright t_j$ in (3) is false. W.l.o.g., we can choose a fact $v_i \xrightarrow{\triangleright} v_j$ (e.g., e_6 in Fig. 9) such that the following conditions hold:
   - $v_i \xrightarrow{\triangleright} v_j$ corresponds to an atom $t_i \triangleright t_j$ in (3) which is false.
   - The G-premises of the F-premise of $v_i \xrightarrow{\triangleright} v_j$ comprise only of facts corresponding to atoms in (3) that are true.

   Then, using the same induction argument as above, we can show that the F-premise P for the G-coloured fact $v_i \xrightarrow{\triangleright} v_j$ holds. Therefore, the conjunct corresponding to $\langle P, v_i \xrightarrow{\triangleright} v_j \rangle$ is true.

2. $G \wedge I \models \bot$. Assume that the formulæ ($2a$) and (3) hold, i.e., G does not contradict (3). Since ($2a$) corresponds to $\mathcal{I}$ and (3) to

$$\{v_i \to v_j \,|\langle C, v_i \to v_j\rangle \in \mathcal{J}\}\,,$$

$G \wedge I$ must be contradictory (condition 2 in Def. 6).
Otherwise, in order for G to contradict (3), at least one of the atoms in $\{t_n \rhd t_m \,|\, \langle P, v_n \xrightarrow{\rhd} v_m\rangle \in \mathcal{J}\}$ must be false. Using induction, we show that the condition of $v_n \xrightarrow{\rhd} v_m$ holds, contradicting the assumption that $\neg(t_n \rhd t_m)$ holds. Thanks to condition (1b) in Def. 6, the F-premise of $v_n \xrightarrow{\rhd} v_m$ is a subset of $\mathcal{I}$. It remains to show that the G-condition of $v_n \xrightarrow{\rhd} v_m$ holds.
   - *Base case.* The height of the tree-shaped derivation of $v_i \xrightarrow{\rhd} v_j$ is one; Thus the G-condition of $v_i \xrightarrow{\rhd} v_j$ is a subset of the atoms in G.
   - *Hypothesis.* The G-condition of $v_i \xrightarrow{\rhd} v_j$ is implied by G and Formula ($2a$) if the height of the tree-shaped derivation is $n-1$ or less.
   - *Induction step.* The height of the derivation of $v_i \xrightarrow{\rhd} v_j$ is n. W.l.o.g., pick a fact $v_n \to v_m$ from the G-condition of $v_i \xrightarrow{\rhd} v_j$. The height of the tree-shaped derivation of this fact is $n-1$ or less. The F-condition for $v_n \to v_m$ holds because of Def. 5, condition (1a) in Def. 6, and the assumption that Formula ($2a$) holds. The G-condition of $v_n \to v_m$ holds by our induction hypothesis, and therefore $v_n \to v_m$ and the G-condition of $v_i \xrightarrow{\rhd} v_j$ must hold.
3. Condition 3 in Def. 6 and the fact that we simplify terms $t_i \rhd t_i$ to **true** or **false** guarantee that I refers only to shared variables and function symbols.

■

The next section discusses applications of our interpolating decision procedure and provides an evaluation of its adequacy for verifying systems software.

5 Application and Evaluation

The two most prominent interpolation-based software model checking techniques are predicate abstraction [11] and interpolation-based abstraction [6]. Both techniques construct an abstract reachability tree by unwinding the (abstract) transition relation. The nodes in this tree are labelled with interpolants derived from infeasible counterexamples (i.e., unsatisfiable conjunctions of relations), thus over-approximating the set of safely reachable program states. The verification process terminates if a fixed-point of this set is reached.

The transition function of programs is typically represented using first order logic formulæ. The primitive data-types of a vast majority of programming languages have bounded domains. In order to be able to apply interpolation-based techniques in a *sound* manner, the decision procedure must not conclude that a formula is unsatisfiable if it is satisfiable in its bit-vector interpretation. This is not guaranteed if we use linear arithmetic over $\mathbb{R}$ or $\mathbb{Z}$: The operator $+$ in infinite interpretations is addition on an infinite set, while it corresponds to

addition mod m (for some m) in the case of bit-vectors. Consider the formula $a > b + 2 \wedge a \leq b$ over the 2-bit variables a and b. This formula has the satisfying assignment $\{a \mapsto 2, b \mapsto 2\}$ in its bit-vector interpretation, while it is unsatisfiable in the theory of linear arithmetic over the reals or the integers.

While the ability of our algorithm to handle arithmetic operations is very limited (our rewriting rules can simplify terms involving addition in only certain special cases), it does not falsely conclude unsatisfiablity for the bit-vector interpretation. However, we may fail to prove unsatisfiability in certain cases (for instance, a chain of 2^n disequalities over $2^n + 1$ distinct n-bit variables). The reason underlying this problem is that the Nelson-Oppen method requires theories to be stably infinite, which is not the case for the theory of bit-vectors. This may lead to spurious counterexamples, which can be caught by falling back to a bit-level accurate decision procedure (such as bit-flattening [12]).

Finally, we have to ask whether our logic is *sufficient* to represent the transition relation of realistic programs. Whether the relations and interpreted functions provided by $\mathcal{L}$ are sufficient depends largely on the application domain. A common benchmark for software model checking tools is the set of Windows device drivers used in [6]. In order to evaluate the usefulness of our logic $\mathcal{L}$, we have integrated the decision procedure into our prototypical interpolation-based model checker Wolverine. Wolverine is an implementation of the algorithm presented in [6]. It generates conjunctive formulæ by unwinding the program and labelling the edges in the reachability graph with transition relations. In this setting, formulæ corresponding to infeasible paths are unsatisfiable. We ran Wolverine on the `kbfiltr.i`, `floppy.i`, and `mouclass.i` drivers presented in [13,6]. Our decision procedure was able to provide interpolants for all unsatisfiable formulæ encountered during the verification process.[5] We attribute this to the fact that device drivers make little use of arithmetic. The loops typically iterate over initialised induction variables, which can be handled by constant propagation (resulting in ground terms that can be rewritten).

6 Related Work

The related work in the area of decision procedures is vast. We focus on recent *interpolating* decision procedures. The first implementation of an interpolating decision procedure widely used in verification is McMillan's FOCI [3]. This tool supports linear arithmetic over $\mathbb{R}$ and equality with uninterpreted functions (EUF), and introduces the semantic discrepancy discussed in Section 5 when used for program verification. Based on the ideas in [3], Fuchs presents a graph-based approach for EUF [4]. The interpolants in CNF generated by this technique are reported to be (syntactically) smaller than the results of FOCI. In comparison, we support a strict super-set of EUF and generate interpolants in DNF. Fuchs' work has recently been extended to combined theories [5], and our algorithm can be seen as an instance of that framework. An interpolating

[5] We do not present results on the run-time, as the performance of Wolverine is not yet competitive due to a lack of optimisation of the model checking algorithm.

decision procedure for the theory of unit-to-variable-per-inequality ($\mathcal{UTVPI}\star$), a logic with atoms of the form $(0 \leq ax_1 + bx_2 + k)$ over $\mathbb{Z}$, is presented in [14]. Jain et al. present an interpolating decision procedure for linear modular equations [15], but does not support uninterpreted functions. We plan to integrate this algorithm into our implementation.

Our algorithm can also be implemented in a Nelson-Oppen or SMT framework, and interpolants can be generated using the mechanisms presented in [10] or [5,16]. It can also be integrated in a *proof-lifting* decision procedure, which constructs word-level proofs from propositional resolution proofs [12].

7 Conclusion and Future Work

We present a decision procedure for a first-order logic fragment with the relations $=$, $\neq$, $\geq$, and $>$ and argue that this logic is an efficiently decidable subset of first order logic. Furthermore, the logic is sound with respect to reasoning about software with bounded integers. We intend to perform an evaluation of a larger scale than presented in this paper. Furthermore, we plan to integrate acceleration techniques similar to [17] into our interpolation-based model checker Wolverine.

Acknowledgements. We thank Philipp Rümmer and May Chan for their detailed comments on our paper and Mitra Purandare for the inspiring discussions.

References

1. McMillan, K.L.: Applications of Craig interpolation to model checking. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 22–23. Springer, Heidelberg (2004)
2. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. The Journal of Symbolic Logic 62, 981–998 (1997)
3. McMillan, K.L.: An interpolating theorem prover. Theoretical Computer Science 345, 101–121 (2005)
4. Fuchs, A., Goel, A., Grundy, J., Krstić, S., Tinelli, C.: Ground interpolation for the theory of equality. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 413–427. Springer, Heidelberg (2009)
5. Goel, A., Krstić, S., Tinelli, C.: Ground interpolation for combined theories. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 183–198. Springer, Heidelberg (2009)
6. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
7. Cimatti, A., Sebastiani, R.: Building efficient decision procedures on top of SAT solvers. In: Bernardo, M., Cimatti, A. (eds.) SFM 2006. LNCS, vol. 3965, pp. 144–175. Springer, Heidelberg (2006)
8. Meir, O., Strichman, O.: Yet another decision procedure for equality logic. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 307–320. Springer, Heidelberg (2005)
9. Nieuwenhuis, R., Oliveras, A.: Proof-Producing Congruence Closure. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 453–468. Springer, Heidelberg (2005)

10. Yorsh, G., Musuvathi, M.: A combination method for generating interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 353–368. Springer, Heidelberg (2005)
11. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Principles of Programming Languages, pp. 232–244. ACM, New York (2004)
12. Kroening, D., Weissenbacher, G.: Lifting Propositional Interpolants to the Word-Level. In: Formal Methods in Computer-Aided Design, pp. 85–89. IEEE, Los Alamitos (2007)
13. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Principles of Programming Languages, pp. 58–70. ACM, New York (2002)
14. Cimatti, A., Griggio, A., Sebastiani, R.: Interpolant generation for UTVPI⋆. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 167–182. Springer, Heidelberg (2009)
15. Jain, H., Clarke, E.M., Grumberg, O.: Efficient craig interpolation for linear diophantine (dis)equations and linear modular equations. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 254–267. Springer, Heidelberg (2008)
16. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: Delayed theory combination vs. Nelson-Oppen for satisfiability modulo theories: A comparative analysis. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 527–541. Springer, Heidelberg (2006)
17. Kroening, D., Weissenbacher, G.: Verification and falsification of programs with loops using predicate abstraction. Formal Aspects of Computing (2009); (published Online First^{TM})