Artificial Intelligence 257 (2018) 127–157

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Parallelizing SMT solving: Lazy decomposition and conciliation

Xi Cheng [a], Min Zhou [a,*], Xiaoyu Song [b], Ming Gu [a], Jiaguang Sun [a]

[a] *School of Software, TNLIST, KLISS, Tsinghua University, Beijing, China*
[b] *Electrical and Computer Engineering, Portland State University, Portland, OR, USA*

A R T I C L E I N F O

Article history:
Received 1 July 2015
Received in revised form 11 December 2017
Accepted 15 January 2018
Available online 31 January 2018

Keywords:
Satisfiability Modulo Theories
Parallelization
Lazy decomposition
Conciliation
Theory of equality with uninterpreted functions

A B S T R A C T

Satisfiability Modulo Theories (SMT) is the satisfiability problem for first-order formulae with respect to background theories. SMT extends the propositional satisfiability by introducing various underlying theories. To improve the efficiency of SMT solving, many efforts have been made on low-level algorithms but they generally cannot leverage the capability of parallel hardware. We propose a high-level and flexible framework, namely lazy decomposition and conciliation (LDC), to parallelize solving for quantifier-free SMT problems. Overall, a SMT problem is firstly decomposed into subproblems, then local reasoning inside each subproblem is conciliated with the global reasoning over the shared symbols across subproblems in parallel. LDC can be built on any existing solver without tuning its internal implementation, and is flexible as it is applicable to various underlying theories. We instantiate LDC in the theory of equality with uninterpreted functions, and implement a parallel solver PZ3 based on Z3. Experiment results on the `QF_UF` benchmarks from SMT-LIB as well as random problems show the potential of LDC, as (1) PZ3 generally outperforms Z3 in 4 out of 8 problem subcategories under various core configurations; (2) PZ3 usually achieves super-linear speed-up over Z3 on problems with sparse structures, which makes it possible to choose an appropriate solver from Z3 and PZ3 in advance according to the structure of input problem; (3) compared to PCVC4, a state-of-the-art portfolio-based parallel SMT solver, PZ3 achieves speed-up on a larger portion of problems and has better overall speed-up ratio.

1. Introduction

Satisfiability is the fundamental problem of determining if a formula can be satisfied by an interpretation. A variety of problems involved with complex constraints, can be described in terms of satisfiability and solved efficiently, such as scheduling [1], design automation [2] and verification [3,4]. Many of these problems are encoded in Boolean formulae and solved by Boolean satisfiability (SAT) solvers. Although SAT problem is proven to be NP-complete [5], modern SAT solvers (zChaff [6], MiniSAT [7], etc.) employ sophisticated and effective heuristics to handle problems of interest efficiently. SAT uses only propositional variables and operations, which leads to expensive encoding for problems natively modeled at a higher level of abstraction. In the past decades, Satisfiability Modulo Theories (SMT) has drawn wide attention where atomic formulae can be not only propositions but also predicates in background theories. Problems requiring expressiveness of equality, uninterpreted function symbols, arithmetic, recursive data structures and quantifiers can be handled naturally and efficiently by SMT solvers.

* Corresponding author.
E-mail address: mzhou@tsinghua.edu.cn (M. Zhou).

https://doi.org/10.1016/j.artint.2018.01.001
0004-3702/© 2018 Elsevier B.V. All rights reserved.

There are mainly two kinds of approaches for SMT solving [8]: the eager and the lazy approaches. The former involves translating an SMT problem into an equi-satisfiable Boolean formula and utilizing the power of optimized SAT solvers. Optimized heuristics focus on generating small SAT problems with relatively fair time expense. The example solvers include UCLID [9], STP [10] and Boolector [11]. The latter approach employs a layered framework where the Boolean skeleton of a formula and the theory related reasoning are separated and handled at different levels. In general the eager translation is ad-hoc and slower than the lazy approach [12]. Most of the state-of-the-art SMT solvers employ lazy approaches, such as Z3 [13], Yices [14], CVC4 [15] and MathSAT [16].

Traditional eager and lazy SMT solvers work in a sequential manner. Efficiency improvement relies on adjusting the algorithm that handles the Boolean skeleton or theory reasoning. However, SMT problem increases its size and complexity tremendously in real-world applications while the gains given by traditional algorithmic improvements fail to effectively respond to this challenge. With the evolution of parallel hardware, it is crucial to parallelize SMT solving to better utilize the capability of hardware.

In this paper, we propose the lazy decomposition and conciliation (LDC) framework for parallel SMT solving on multicore systems. An SMT problem is firstly decomposed into several smaller ones by distributing its clauses, and then solved by conciliating local and global reasoning. SMT solving can benefit from LDC in two aspects. First, the large and complex problem is decomposed into smaller and simpler subproblems that can be solved in parallel. In some cases, satisfiability of the original problem can be directly determined by satisfiability of subproblems. Second, lemmas derived from local reasoning on each subproblem are shared, thus the termination may reach faster than solving the whole problem sequentially. Being a high-level framework, LDC frees solver designers from adjusting low-level algorithm as it requires calling APIs provided by mature sequential solvers only. Moreover, LDC is general and can be applied to various underlying theories. We have instantiated LDC in the theory of equality with uninterpreted functions ($\mathcal{T}_{\mathsf{E}}$), and implemented a parallel solver named PZ3 based on Z3 [13], one of the state-of-the-art SMT solvers. Experiment results show the competitive efficiency of PZ3 over Z3. Particularly, PZ3 has drastic performance gains on sparse SMT problems.

Main contributions of this paper are summarized as follows:

- We propose a high-level and flexible framework LDC for parallelizing SMT solving on multicore systems, as (1) it can be built upon an existing SMT solver in a loosely-coupled manner without modifying complicated internal algorithms; (2) it is applicable to various practical theories. The soundness and completeness of LDC are also formally established.
- We present an instantiation of LDC in $\mathcal{T}_{\mathsf{E}}$. The decision procedure can be proven to terminate within a finite number of steps.
- We implement the instantiation of LDC in $\mathcal{T}_{\mathsf{E}}$ as a prototype tool named PZ3 based on Z3. The experimental results show that PZ3 outperforms Z3 on various kinds of random or crafted problems, especially ones with sparse structures.

The rest of the paper is organized as follows. Section 2 introduces formal preliminaries. In Section 3, we illustrate our framework by a motivating example. Section 4 presents LDC framework and Section 5 instantiates LDC in $\mathcal{T}_{\mathsf{E}}$. In Section 6, we prove some important properties of LDC and its instantiation in $\mathcal{T}_{\mathsf{E}}$. Section 7 demonstrates a detailed experimental evaluation of our parallelized solver PZ3. Finally, we survey some related work in Section 8 and conclude the paper in Section 9.

2. Preliminaries

In this section, we introduce basic concepts and notations used in our paper.

2.1. First-order logic

The *signature* Σ of a first-order language is a tuple (Σ^F, Σ^P) where Σ^F and Σ^P are the sets of function and predicate symbols, respectively. Each symbol in Σ is associated with an arity, a non-negative number. In particular, we call 0-arity symbols in Σ^F constant symbols, and 0-arity symbols in Σ^P propositional symbols. A variable is a symbol denoting an arbitrary object. A Σ-*term* is a first-order term constructed using symbols in Σ. A Σ-*atom* is either an expression of the form $P(t_1, \ldots, t_n)$ where $P \in \Sigma^P$ and $t_1, \ldots, t_n$ are Σ-terms, or an expression of the form $t_1 = t_2$ where $=$ is the logical equality symbol and t_1, t_2 are Σ-terms. A Σ-*literal* is a Σ-atom or its negation. A Σ-*formula* is a Boolean combination of Σ-literals. In particular, a Σ-*clause* is a disjunction of Σ-literals. Σ-*sentences* are Σ-formulae without free variables. In this paper we are interested in quantifier-free terms and formulae. For technical convenience, we treat variables in quantifier-free formula as constants in a suitable expansion of Σ.

A *model* (or an *interpretation*) M is a pair $(\mathcal{D}, (\_)^M)$ where $\mathcal{D}$ is a non-empty set and $(\_)^M$ is a map from symbols in Σ to concrete values in $\mathcal{D}$. $(\_)^M$ assigns each constant symbol $c \in \Sigma^F$ to an element $c^M \in \mathcal{D}$; each function symbol f of arity $n > 0$ to a total function $f^M : \mathcal{D}^n \to \mathcal{D}$; each propositional symbol p to an element $p^M \in \{\mathsf{true}, \mathsf{false}\}$ and each predicate P of arity $n > 0$ to a total function $P^M : \mathcal{D}^n \to \{\mathsf{true}, \mathsf{false}\}$. In the rest of this paper, we use the term "interpretation" instead of "model" because a "model" is defined over the whole Σ while an "interpretation" can be a valuation on a subset of symbols. The definition of interpretation can be extended naturally to terms and formulae. For a term t (or a formula ϕ), we denote

t^M (or ϕ^M) the value of t (or ϕ) under the interpretation M. We say M *satisfies* (resp. *falsifies*) a formula ϕ if and only if ϕ^M is true (resp. false).

A Σ-*theory* $\mathcal{T}$ is defined by a set of Σ-sentences. A $\mathcal{T}$-interpretation M is a Σ-interpretation such that all Σ-sentences in $\mathcal{T}$ are evaluated to be true under M. Given a theory $\mathcal{T}$, a formula ϕ is (1) $\mathcal{T}$-valid ($\models_{\mathcal{T}} \phi$) iff ϕ is satisfied by all $\mathcal{T}$-interpretations; (2) $\mathcal{T}$-satisfiable iff there is at least one $\mathcal{T}$-interpretation that satisfies ϕ; (3) $\mathcal{T}$-unsatisfiable iff ϕ is falsified by all $\mathcal{T}$-interpretations.

2.2. The theory of equality

Equality over terms is expressed in the theory of equality, denoted by $\mathcal{T}_\mathsf{E}$. The signature of $\mathcal{T}_\mathsf{E}$ consists of $\Sigma^F = \{f, g, h, \dots\}$ and $\Sigma^P = \{=, P, Q, R, \dots\}$. $\mathcal{T}_\mathsf{E}$ has four axioms: reflexivity, symmetry, transitivity and consistency. The first three axioms define the predicate "$=$" as an equivalence relation, while the last axiom guarantees that each input of a function or predicate is associated to exactly one output. All four axioms together make "$=$" a *congruence relation*. The other symbols in the signature are *uninterpreted* as $\mathcal{T}_\mathsf{E}$ does not impose any assumptions on the their interpretations except for the requirement of consistency. Since we are mostly interested in uninterpreted symbols, we use the term "symbol" to refer to uninterpreted symbols in the rest of the paper.

Though the validity problem for $\mathcal{T}_\mathsf{E}$ is undecidable due to the undecidability of first-order logic [17], its quantifier-free fragment is decidable in polynomial time using a procedure known as congruence closure [18]. Given a formula ϕ, this algorithm attempts to construct a congruence relation over subterms of ϕ, or to prove that such relation does not exist.

Example 1. Consider a formula:

$$\phi : f(a) = f(b) \wedge a \neq b$$

Subterms of ϕ are $S_\phi = \{a, b, f(a), f(b)\}$. By the congruence closure algorithm, we can construct a relation R of subterms given by the partition $S_\phi/R = \{\{a\}, \{b\}, \{f(a), f(b)\}\}$. Therefore, ϕ is $\mathcal{T}_\mathsf{E}$-satisfiable.

2.3. Craig interpolation

A Craig interpolant ρ is a formula for an inconsistent pair of formulae (ϕ, ψ) such that ρ is implied by ϕ, inconsistent with ψ and refers only to non-logical symbols occurring in both ϕ and ψ.[1] By Craig's interpolation theorem [19], such interpolant ρ exists for an arbitrary inconsistent pair of formulae (ϕ, ψ) in first-order logic. Interpolation was proposed for model checking, firstly in propositional logic [20]. Craig interpolation also holds for underlying theories [21]. Suppose an inconsistent pair of formulae (ϕ, ψ) in theory $\mathcal{T}$, there exists an interpolant ρ such that (1) $\phi \rightarrow_{\mathcal{T}} \rho$, which denotes that $\phi \rightarrow \rho$ holds under all the $\mathcal{T}$-interpretations; (2) ρ is inconsistent with ψ; (3) ρ refers only to non-logical symbols of ϕ and ψ in common. Eligible theories include equality [22,23], linear real arithmetic [22,24], Presburger arithmetic [24,25], arrays without extensionality [24,26,27], etc.

There are two main kinds of approaches to compute interpolant. The first extracts interpolant from the proof derived from inconsistency [20,22,28], which is adopted by some important interpolating theorem provers such as Z3 [29] and MathSAT [16]. The latter involves reduction of the interpolation problem to constraint solving [23,30].

3. A motivating example

In this section we demonstrate our parallel SMT solving technique informally using a small example. Consider the following unsatisfiable $\mathcal{T}_\mathsf{E}$-formula ϕ:

$$\begin{aligned}
&((x_0 = y_0) \vee (x_0 = z_0)) \wedge ((y_0 = x_1) \vee (x_0 = z_0)) \wedge \\
&((x_0 = y_0) \vee (z_0 = x_1)) \wedge ((y_0 = x_1) \vee (z_0 = x_1)) \wedge \\
&((x_1 = y_1) \vee (x_1 = z_1)) \wedge ((y_1 = x_2) \vee (x_1 = z_1)) \wedge \\
&((x_1 = y_1) \vee (z_1 = x_2)) \wedge ((y_1 = x_2) \vee (z_1 = x_2)) \wedge \\
&((x_2 = y_2) \vee (x_2 = z_2)) \wedge ((y_2 = x_3) \vee (x_2 = z_2)) \wedge \\
&((x_2 = y_2) \vee (z_2 = x_3)) \wedge ((y_2 = x_3) \vee (z_2 = x_3)) \wedge \\
&(x_0 \neq x_3)
\end{aligned}$$

ϕ is in CNF consisting of 13 clauses and contains 10 constant symbols. In what follows, we solve the satisfiability of ϕ using 3 threads $\mathrm{T}_0, \mathrm{T}_1, \mathrm{T}_2$ in parallel.

[1] The original Craig interpolant ρ is for a valid implication $\phi \rightarrow \psi$, such that (1) $\phi \rightarrow \rho$; (2) $\rho \rightarrow \psi$ and (3) ρ is defined over common symbols of ϕ and ψ. The interpolant for an inconsistent pair of formulae is also called *reverse interpolant*. However, this does not make a substantial difference in the context of this paper.

Table 1
Distribution of symbols in subformulae ψ_1 and ψ_2. • in column x_0 and row ψ_1 indicates that x_0 appears in ψ_1.

	x_0	y_0	z_0	x_1	y_1	z_1	x_2	y_2	z_2	x_3
ψ_1	•	•	•	•	•	•	•			
ψ_2	•						•	•	•	•
shared	•						•			

Decomposition We distribute clauses of ϕ into 2 subformulae ψ_1, ψ_2. The first 8 clauses are distributed to ψ_1 and the others are distributed to ψ_2. Table 1 shows that ψ_1 contains 7 symbols and ψ_2 contains 5 symbols, while they share only 2 symbols.

Solving subformulae We check the satisfiability of ψ_1 on thread T_1 and ψ_2 on thread T_2. Both ψ_1 and ψ_2 are satisfiable, however, it is insufficient to conclude that $\psi_1 \wedge \psi_2$ is satisfiable because they may have inconsistent interpretations on the shared symbols. Then we compute an interpretation on shared symbols (namely *shared interpretation*) on the thread T_0 such that this interpretation is consistent with ψ_1 and ψ_2 simultaneously.

Conciliation We choose a candidate shared interpretation M_S arbitrarily, for example, one that corresponds to the congruence relation $\{\{x_0\}, \{x_2\}\}$. M_S can be characterized as the constraint $F_S : x_0 \neq x_2$. Next we check whether F_S is consistent with ψ_1 and ψ_2 on threads T_1, T_2, respectively. It is found that (ψ_1, F_S) is inconsistent while (ψ_2, F_S) is consistent. Then an interpolant $i_{11} : x_2 = x_0$ is derived from (ψ_1, F_S). By definition, we have $\psi_1 \rightarrow_{\mathcal{T}_E} i_{11}$, thus i_{11} is one of the logical consequences implied by ϕ. In other words, interpolants can guide the search for a feasible shared interpretation.

Then, we choose another shared interpretation M'_S satisfying the constraint i_{11} and it corresponds to the congruence relation $\{\{x_0, x_2\}\}$ which is characterized as the formula $F'_S : x_0 = x_2$. From the inconsistency of (ψ_2, F'_S), we derive an interpolant $i_{22} : x_2 \neq x_0$.

Next, we search for a shared interpretation M''_S which satisfies i_{11} and i_{22}. However, such interpretation does not exist because i_{11} and i_{22} are in conflict. Formally, we have $\phi \rightarrow_{\mathcal{T}_E} (i_{11} \wedge i_{22}) \rightarrow_{\mathcal{T}_E} \perp$ which implies that ϕ is falsified by all the $\mathcal{T}_E$-interpretations. In other words, ϕ is unsatisfiable.

This example shows that our parallel SMT solving framework can be advantageous over the sequential DPLL($\mathcal{T}$) in two aspects: (1) the search space for interpretations is substantially reduced, as in this running example, the interpretation space for the original problem involves 10 constants while the space for shared interpretations involves only 2, thus the conflicts could be discovered faster, (2) parallel threads share derived interpolants, which enables multiple threads to prune the search space in different manners simultaneously.

4. The LDC framework

The LDC framework encompasses two main steps: *lazy decomposition* and *conciliation*. First the input formula ϕ is decomposed into subformulae. If subformulae do not share symbols or any of them is unsatisfiable, (un)satisfiability of ϕ can be directly derived. Otherwise, local reasoning inside subformulae is conciliated by unifying their interpretations on shared symbols. Finally, we can either find an interpretation on shared symbols which is a witness of satisfiability of ϕ, or prove that such interpretation does not exist (thus ϕ is unsatisfiable). The complete procedure of LDC framework is shown in Algorithm 1.

4.1. Lazy decomposition

Let $\phi = \phi_1 \wedge \cdots \wedge \phi_n$ be in CNF. A *lazy decomposition* $\psi = \{\psi_1, \ldots, \psi_k\}$ is a partition of $\{\phi_1, \ldots, \phi_n\}$ into k disjoint subsets of clauses. Thus, $\psi_1, \ldots, \psi_k$ are k subformulae of ϕ. A symbol x is *shared* in decomposition ψ if it is contained in multiple subformulae. We use Σ_ψ to denote the set of all shared symbols in ψ. If Σ_ψ is empty, the satisfiability of ϕ is trivially implied (shown from line 2 to 6). Otherwise, the satisfiability of $\psi_1, \ldots, \psi_k$ can not imply the satisfiability of ϕ because interpretations of shared symbols by subformulae may be inconsistent. Our decomposition method is lazy because shared symbols are handled after solving each subformula. Laziness avoids expensive computations for resolving overlapping of subformulae directly on the formula level.

4.2. Conciliation

The *conciliation* step determines whether a global interpretation satisfying ϕ exists by searching for an interpretation M_S on shared symbols where M_S can be extended to a satisfying global interpretation. This step consists of iterations across lines 10–21. G denotes *global invariant*, which is a conjunction of formulae implied by ϕ. Before the first iteration, we pick an arbitrary shared interpretation M_S since G is true (and thus there is no constraint on M_S). In each iteration, we firstly characterize M_S as a semantically equivalent formula F_S, and then check the consistency of F_S with each subformula ψ_i by `Interpolate`, which returns an interpretation satisfying $\psi_i \wedge F_S$ if (ψ_i, F_S) is consistent, or an interpolant I_i such that

Algorithm 1: The LDC framework.

```
   Input  : $\phi$, $k$
   Output : sat if $\phi$ is satisfiable; unsat otherwise
1  if $k \le 1$ then return Solve($\phi$);
2  $\{\psi_1, \dots, \psi_k\} \leftarrow$ Decompose($\phi$,$k$);
3  parfor $i = 1, \dots, k$ do
4      $s_i \leftarrow$ Solve($\psi_i$);
5      if $s_i =$ unsat then return unsat;
6  if $\Sigma_\psi = \emptyset$ then return sat;
7  $G \leftarrow$ true;
8  $M_S \leftarrow$ ComputeSharedInterp($G$,$\psi$);
9  repeat
10     $F_S \leftarrow$ Formulate($M_S$);
11     $G' \leftarrow G$;
12     parfor $i = 1, \dots, k$ do
13         $(I_i, M_i, flag) \leftarrow$ Interpolate($\psi_i$,$F_S$);
14         if $flag =$ true then $G' \leftarrow G' \wedge I_i$;
15     if $G' = G$ then
16         if CombineInterp($M_S, M_1, \dots, M_k$) =true then return sat;
17         else $M_S \leftarrow$ RefineSharedInterp($M_S, M_1, \dots, M_k$);
18     else
19         $G \leftarrow G'$;
20         if $G$ is unsatisfiable then return unsat;
21         else $M_S \leftarrow$ ComputeSharedInterp($G$,$\psi$);
22 until false;
```

$\psi_i \to I_i \to \neg F_S$ otherwise. The interpolant I_i can be regarded as an over-approximation of ψ_i with respect to F_S, and it is used to refine the global invariant G, as shown in line 14. Even if k pairs $(\psi_1, F_S), \dots, (\psi_k, F_S)$ are all consistent, ϕ is not necessarily satisfiable because $M_1, \dots, M_k$ may not be combinable into a global interpretation for ϕ. The following example demonstrates this point.

Example 2. Consider the unsatisfiable formula ϕ in $\mathcal{T}_\mathsf{E}$ along with the decomposition $\psi = \{\psi_1, \psi_2\}$:

$$\phi = \underbrace{f(a) = b \wedge f(b) \neq c \wedge a = c}_{\psi_1} \wedge \underbrace{f(a) = d \wedge f(d) = e \wedge a = e}_{\psi_2}$$

By definition $\Sigma_\psi = \{f, a\}$. We arbitrarily choose an interpretation M_S on Σ_ψ corresponding to the congruence relation $\{\{a\}, \{f(a)\}\}$. M_S is characterized as the formula $F_S : f(a) \neq a$, which is consistent with both ψ_1 and ψ_2. However, M_S is not the witness of satisfiability because ϕ is unsatisfiable.

Thus, we check whether $M_1, \dots, M_k$ are combinable by `CombineInterp` (line 16). If they are combinable, then ϕ is satisfiable as the combination of $M_1, \dots, M_k$ is an interpretation that satisfies ϕ; otherwise M_S needs to be refined (line 17) for the next iteration.

If there exists an inconsistent pair (ψ_i, F_S), the global invariant G will be refined by new derived interpolants. Then we compute the new shared interpretation (line 21) for the next iteration by checking the satisfiability of G. If G is unsatisfiable, ϕ is concluded to be unsatisfiable (line 20).

The conciliation step does not guarantee termination in general. However, for some theories such as $\mathcal{T}_\mathsf{E}$ we can instantiate it with the certain termination property.

5. Instantiation in $\mathcal{T}_\mathsf{E}$

To instantiate the LDC framework, one needs to carefully design the instantiations of 7 interfaces: `Decompose`, `Solve`, `ComputeSharedInterp`, `Formulate`, `Interpolate`, `CombineInterp` and `RefineSharedInterp`.

5.1. `Decompose`: *make lazy decomposition*

`Decompose` distributes the clauses of the input formula to generate subformulae. The selection of decomposition heuristic could essentially influence the efficiency of the decision procedure since the decomposition schema determines the number of shared symbols and the complexities of subproblems. A desired decomposition ψ should have Σ_ψ of small size, because the conciliation step could terminate faster.

Algorithm 2: The greedy decomposition algorithm.

```
   Input  : $\phi$,$k$
   Output : $\{\psi_1,\dots,\psi_k\}$
1  construct $G^\phi$ along with the weight function $w^\phi$;
2  $S \leftarrow \emptyset$;
3  foreach $i = 1,\dots,k-1$ do
4      $\psi_i \leftarrow$ true;
5      if $V^\phi = \emptyset$ then continue;
6      find $v^* \in V^\phi$ with the minimum $|\Sigma_{v^*}|$;
7      remove $v^*$ from $G^\phi$;
8      $\psi_i \leftarrow \psi_i \wedge v^*$;
9      $S \leftarrow S \cup \Sigma_{v^*}$;
10     foreach $v_1, v_2 \in V^\phi$ do
11         $w^\phi((v_1,v_2)) \leftarrow (\Sigma_{v_1} \setminus S) \cap (\Sigma_{v_2} \setminus S)$;
12     foreach $v' \in V^\phi$ such that $w^\phi((v',\_)) = \emptyset$ do
13         remove $v'$ from $G^\phi$;
14         $\psi_i \leftarrow \psi_i \wedge v'$;
15     $S \leftarrow S \cup \Sigma_{\psi_i}$;
16 $\psi_k \leftarrow \bigwedge_{v \in V^\phi} v$;
17 return $\{\psi_1,\dots,\psi_k\}$
```

Decomposition can be modeled as the graph partition problem. The *clause graph* of formula ϕ is $G^\phi = (V^\phi, E^\phi)$, where the set of vertices V^ϕ corresponds to the set of clauses $\{\phi_1,\dots,\phi_n\}$ and the set of edges $E^\phi \subseteq V^\phi \times V^\phi \times 2^\Sigma$ consists of labeled edges (v_1, v_2, S) where the clauses v_1 and v_2 share the set of symbols S. Formally, the label function of edges $w^\phi : E^\phi \to \Sigma$ is defined as

$$w^\phi((v_1,v_2)) = \Sigma_{v_1} \cap \Sigma_{v_2}$$

where Σ_{v_i} denotes the set of symbols in the clause v_i $(i = 1, 2)$. To minimize the size of Σ_ψ, it suffices to find $\{V_1^\phi,\dots,V_k^\phi\}$, a k-cut of V^ϕ that minimizes

$$\left| \bigcup_{i=1}^{k} \bigcup_{j=i+1}^{k} \bigcup_{v_1 \in V_i^\phi, v_2 \in V_j^\phi} w^\phi((v_1,v_2)) \right|$$

Finding the exact solution of this combinatorial optimization problem is generally NP-hard and computationally prohibitive in practice, especially when the size of ϕ is very large. Therefore a fast greedy approximation algorithm is employed instead.

Algorithm 2 illustrates our greedy decomposition. To decompose the input formula into k subformulae ($k > 1$), the algorithm runs in $(k-1)$ iterations. In each iteration, we find a clause v^* from V^ϕ with the minimum set of symbols, remove it from G^ϕ and distribute it to the current subformula ψ_i. Next, we add symbols in distributed clause to the set S, and update the weight function w^ϕ by eliminating symbols in S. Then all the isolated clauses (a clause $v \in V^\phi$ is isolated when $w^\phi((v, v')) = \emptyset$ for any other clause $v' \in V^\phi$) are distributed to ψ_i. All the symbols contained in ψ_i are added to S for the next iteration. After $(k-1)$ iterations, $(k-1)$ subformulae are generated and the remaining clauses form ψ_k. When V^ϕ becomes empty, the remaining unassigned subformulae are true.

Example 3. Decompose the following formula ϕ into 3 subformulae.

$$\underbrace{(a \vee b \vee \neg c)}_{v_1} \wedge \underbrace{(c \vee d \vee e)}_{v_2} \wedge \underbrace{(a \vee c)}_{v_3} \wedge \underbrace{(c \vee \neg e)}_{v_4}$$

First we construct the clause graph G^ϕ for input formula ϕ, shown in Fig. 1a. Notice that clause v_3 contains the fewest symbols, then we remove v_3 from G^ϕ and update weight function, shown in Fig. 1b. Now v_1 is an isolated node as it shares no symbols with any other clauses. Thus we remove v_1 and update the G^ϕ again. After the first iteration we have $\psi_1 : v_1 \wedge v_3$.

Next we work on the clause graph shown in Fig. 1c to yield the second subformula $\psi_2 : v_4$. Finally, the remaining clause v_2 composes the third subformula ψ_3. The output decomposition is $\psi = \{\psi_1, \psi_2, \psi_3\}$ where $\Sigma_\psi = \{c, e\}$.

The time complexity of Algorithm 2 is discussed as follows. First the construction of G^ϕ has $\mathcal{O}(|V^\phi|^2)$. For one iteration from line 3 to 15, finding v^* (in line 6) takes $\mathcal{O}(|V^\phi|)$ time, updating weight function (from line 10 to 11) takes $\mathcal{O}(|V^\phi|^2)$ time and distributing clauses for ψ_i (from line 12 to 14) has $\mathcal{O}(|V^\phi|^2)$ time complexity. Hence, the overall worst-case time complexity of Algorithm 2 is $\mathcal{O}(k|V^\phi|^2)$ which is polynomial in the number of clauses of the input formula ϕ.

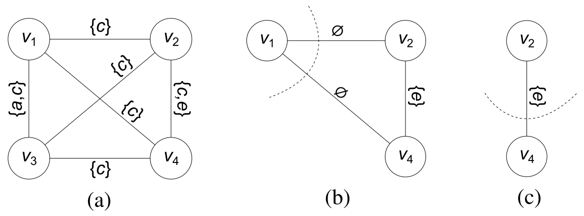


Fig. 1. The procedure of greedy decomposition method on formula ϕ in Example 3.

Algorithm 3: The procedure of `ComputeSharedInterp`.

```
Input  : $G$,$\psi$
Output : $M_S$
1 if $G =$ true then
2     $M_S \leftarrow (\mathcal{D}_S, (\_)^{M_S})$ where $\mathcal{D}_S = \{\mu_1, \ldots, \mu_m\}$, $c_i^{M_S} = \mu_i$ for each constant $c_i \in \Sigma_\psi$ $(1 \le i \le m)$;
3     return $M_S$
4 compute $M_S$ such that $M_S \models G$;
5 foreach constant $c \in \Sigma_\psi$ and $c^{M_S}$ is not defined do
6     $(\_)^{M_S} \leftarrow (\_)^{M_S} \cup (c, \mu)$ where $\mu \in \mathcal{D}_S$;
7 return $M_S$
```

5.2. `Solve` *and* `Interpolate`

`Solve` is to determine the satisfiability of a subformula by invoking an existing solver. `Interpolate` attempts to compute the interpolant between two formulae ψ_i and F_S. If they are inconsistent, we compute an interpolant I_i and set $flag$ to **true**; otherwise we derive an interpretation M_i satisfying $\psi_i \wedge F_S$ and set $flag$ to **false**. Existing interpolation engine, such as Z3, MathSAT, can be utilized to compute interpolant in various theories.

5.3. `ComputeSharedInterp`: *compute a shared interpretation*

`ComputeSharedInterp` is to derive a shared interpretation satisfying the global invariant G. In the following, we define some concepts to serve as the vehicle for formalizing our discussion.

Definition 1 *($\mathcal{T}_\mathsf{E}$-interpretation).* Given a congruence relation $\{C_1, \ldots, C_n\}$ where $C_1, \ldots, C_n$ are n congruence classes of terms, there is a $\mathcal{T}_\mathsf{E}$-interpretation $M = (\mathcal{D}, (\_)^M)$ such that

1. $\mathcal{D} = \{\mu_1, \ldots, \mu_n\}$ where $\mu_1, \ldots, \mu_n$ are labels for $C_1, \ldots, C_n$ respectively;
2. $c^M = \mu_i$ if c is a constant symbol and $c \in C_i$;
3. $f^M(\mu_{i_1}, \ldots, \mu_{i_k}) = \mu_j$ if there exists a term $f(\bar{t_1}, \ldots, \bar{t_k}) \in C_j$ and $\bar{t_1} \in C_{i_1}, \ldots, \bar{t_k} \in C_{i_k}$.

It is possible to assign a different set of labels as the domain of a $\mathcal{T}_\mathsf{E}$-interpretation. Thus, a congruence relation can be represented by many *isomorphic* interpretations. The isomorphism over $\mathcal{T}_\mathsf{E}$-interpretations is defined below.

Definition 2 *(Isomorphism of $\mathcal{T}_\mathsf{E}$-interpretations).* Let M_1 and M_2 be $\mathcal{T}_\mathsf{E}$-interpretations, $\mathcal{D}_1$ and $\mathcal{D}_2$ be the domains of M_1 and M_2 respectively. A map $h : \mathcal{D}_1 \to \mathcal{D}_2$ is an isomorphism of M_1 into M_2 if the following condition holds:

1. h is bijective;
2. $h(c^{M_1}) = c^{M_2}$ for each constant symbol $c \in \Sigma$;
3. $h(f^{M_1}(d_1, \ldots, d_n)) = f^{M_2}(h(d_1), \ldots, h(d_n))$, for each n-ary function symbol $f \in \Sigma$ and $d_1, \ldots, d_n \in \mathcal{D}_1$.

We use $M_1 \cong M_2$ to denote that M_1 is isomorphic to M_2.

Next we give a formal definition of shared interpretation.

Definition 3 *(Shared interpretation).* Given a formula ϕ and its decomposition $\psi = \{\psi_1, \ldots, \psi_k\}$, M_S is a shared interpretation if for each constant symbol $c \in \bigcup_{1 \le i < j \le k} (\Sigma_{\psi_i} \cap \Sigma_{\psi_j})$, c^{M_S} is defined.

The procedure of `ComputeSharedInterp` is shown in Algorithm 3. In essence, we derive a shared interpretation M_S by computing an interpretation M satisfying G using an existing SMT solver. For each constant c that is not interpreted

Algorithm 4: The procedure of `ExtractSFA`.

```
Input  : $M_S, M_1, \dots, M_k$
Output : $F$: the set of shared function applications which have no valuations in $M_S$
1 foreach $i = 1, \dots, k$ do
2     $M_i \leftarrow$ Unify($M_i$,$M_S$);
3 $F \leftarrow \emptyset$;
4 foreach $(f, \mathbf{d})$ such that $f^{M_S}(\mathbf{d})$ is not defined and there exists $1 \le i < j \le k$ such that both $f^{M_i}(\mathbf{d})$ and $f^{M_j}(\mathbf{d})$ are defined do
5     $F \leftarrow F \cup \{(f, \mathbf{d})\}$;
6 return $F$
```

by M, we specify an arbitrary value in the domain $\mathcal{D}$ for c to refine M. Finally, the fully-refined M is M_S. It is necessary to address uninterpreted constants in this way because computing the shared interpretation is to complete a candidate interpretation for each ψ_i $(1 \le i \le k)$. The outcome is that either for each ψ_i $(1 \le i \le k)$ we luckily find an interpretation that satisfies each ψ_i while M_S is possibly the witness of global satisfiability, or M_S helps to derive new interpolants which are logical consequences of ϕ.

5.4. `Formulate`: *characterize interpretation as formula*

`Formulate` is to characterize an interpretation M_S as a semantically equivalent formula F_S. In $\mathcal{T}_{\mathsf{E}}$, an interpretation is associated with a congruence relation corresponding to a partition of terms, thus F_S consists of two kinds of clauses: (1) equalities over terms in each congruence class; (2) inequalities over the representatives of all the congruence classes. More specifically, let M_S be an interpretation corresponding to a congruence relation represented as

$$\left\{\{t_1^1, \dots, t_1^{k_1}\}, \dots, \{t_n^1, \dots, t_n^{k_n}\}\right\}$$

To characterize M_S as a formula F_S, follow the steps below:

(1) build n equalities for n congruence classes:

$$\begin{aligned} C_1 &: t_1^1 = \cdots = t_1^{k_1} \\ &\vdots \\ C_n &: t_n^1 = \cdots = t_n^{k_n} \end{aligned}$$

(2) build a constraint expressing the distinction of n representatives:

$$C_{n+1} : \bigwedge_{1 \leqslant i < j \leqslant n} t_i^1 \neq t_j^1$$

(3) conjunct $n+1$ constraints to form F_S:

$$F_S = C_1 \wedge \cdots \wedge C_n \wedge C_{n+1}$$

5.5. `CombineInterp`: *combine interpretations for subformulae*

This procedure checks whether k interpretations $M_1, \dots, M_k$ are combinable under the shared interpretation M_S. Informally, $M_1, \dots, M_k$ can be combined into an interpretation satisfying the input formula ϕ if their valuations on shared symbols are included in M_S. This is necessary when subformulae share function symbols of non-zero arity, as Example 2 illustrates.

Given an interpretation M with the domain $\mathcal{D}$, a function application is a pair $(f, \mathbf{d})$ where f is an n-ary $(n > 0)$ function symbol and $\mathbf{d} \in \mathcal{D}^n$. In the context of LDC, we say $(f, \mathbf{d})$ is a *shared function application* if there exists i, j $(1 \le i < j \le k)$ such that $f^{M_i}(\mathbf{d})$ and $f^{M_j}(\mathbf{d})$ are defined. The `ExtractSFA` procedure, shown in Algorithm 4, is to collect all the shared function applications which have no valuations in M_S. Before collecting function applications, `ExtractSFA` calls the `Unify` to unify each M_i with M_S by substituting the certain domain elements in $\mathcal{D}_i$ of M_i with the domain elements in $\mathcal{D}_S$ of M_S in order to make their interpretations on shared symbols consistent in literal.

The `CombineInterp` procedure, shown in Algorithm 5, is to call the `ExtractSFA` procedure and return whether the returned set F is empty. If F is empty, then all the shared function applications have valuations in M_S and thus $M_1, \dots, M_k$ can be combined into an interpretation satisfying ϕ (in other words, M_S is the witness of global satisfiability).

`Unify` Elements in the domain $\mathcal{D}_i$ of each interpretation M_i are unified with respect to M_S by substitution. The rules for element substitution are shown in Fig. 2. The notation $M[\mu/\nu]$ denotes a new interpretation with all the occurrences of ν in M substituted with μ, where ν is an element in the domain of M. Therefore, the substitution operations make changes

Algorithm 5: The procedure of `CombineInterp`.

```
Input   : $M_S, M_1, \ldots, M_k$
Output  : true if $M_S$ is the witness of global satisfiability; false otherwise
1 $F \leftarrow$ ExtractSFA($M_S, M_1, \ldots, M_k$);
2 return $F = \emptyset$
```

$$\textsc{Const-Sub}\ \frac{c^{M_S} = \mu \qquad c^{M_i} = \nu \qquad \mu \neq \nu}{M_i \leftarrow M_i[\mu/\nu]}$$

$$\textsc{Func-Sub}\ \frac{f^{M_S}(\mu_1, \ldots, \mu_n) = \mu \qquad f^{M_i}(\mu_1, \ldots, \mu_n) = \nu \qquad \mu \neq \nu}{M_i \leftarrow M_i[\mu/\nu]}$$

Fig. 2. The rules of element substitution for `Unify`.

to not only the domain of M, but also $(\_)^M$. The Const-Sub rule substitutes the element ν in $\mathcal{D}_i$ with μ in $\mathcal{D}_S$ if there is a constant symbol c interpreted as μ and ν by M_S and M_i, respectively. The Func-Sub rule is applied to function applications. Fig. 2 shows a special case of this rule for n-ary ($n > 0$) function applications. The element ν in $\mathcal{D}_i$ is substituted as μ in $\mathcal{D}_S$ if $(\mathbf{d}, \nu) \in f^{M_i}$ and $(\mathbf{d}, \mu) \in f^{M_S}$. To unify two interpretations M_i and M_S, we first apply the Const-Sub rule until saturation and then apply the Func-Sub rule until saturation. We have the following lemmas on element substitution.

Lemma 1. *Each element in $\mathcal{D}_i$ can be substituted for at most once.*

Proof. Let Σ_M denote the set of symbols interpreted by M. Consider an element $\nu \in \mathcal{D}_i$ substituted as $\mu \in \mathcal{D}_S$, then there exists a term t which is constituted by the symbols in $(\Sigma_{M_i} \cap \Sigma_{M_S})$, such that $t^{M_i} = \nu$ and $t^{M_S} = \mu$. Let M_i' be the new interpretation after element substitution, we have $t^{M_i'} = \mu$. If μ can be further substituted by $\mu' \in \mathcal{D}_S$ where $\mu \neq \mu'$, then $t^{M_S} = \mu'$. Hence we have $(t = t)^{M_S} = (\mu = \mu') = \mathsf{false}$, which is a contradiction. □

Lemma 2. *Upon termination of the* `Unify` *procedure, none of the substitution rules are applicable.*

Proof. Assume the Const-Sub rule is applicable, then there exists a constant $c \in (\Sigma_{M_i} \cap \Sigma_{M_S})$ such that $c^{M_S} = \mu, c^{M_i} = \nu$ and $\mu \neq \nu$. If ν is the substituted element, then the assumption contradicts to Lemma 1. Otherwise, this element should be substituted before applying the Func-Sub rule. It is trivial that the Func-Sub rule is not applicable when the `Unify` procedure finishes. □

Example 4. Consider the formula ϕ and its decomposition $\psi = \{\psi_1, \psi_2\}$ in Example 2. We arbitrarily choose a shared interpretation M_S which corresponds to the congruence relation $\{\{a\}, \{f(a)\}\}$, and its semantically equivalent formula F_S is $a \neq f(a)$. Let M_1 and M_2 be interpretations such that $M_1 \models (\psi_1 \wedge F_S)$ and $M_2 \models (\psi_2 \wedge F_S)$. The congruence relations of M_1 and M_2 are

$$\begin{aligned} &M_1 : \{\{f(a), b\}, \{f(b)\}, \{a, c\}\} \\ &M_2 : \{\{f(a), d\}, \{a, e, f(d)\}\} \end{aligned}$$

We have $M_S = (\mathcal{D}_S, (\_)^{M_S})$, $M_1 = (\mathcal{D}_1, (\_)^{M_1})$ and $M_2 = (\mathcal{D}_2, (\_)^{M_2})$, such that

$$\begin{aligned} \mathcal{D}_S &= \{\mu_1, \mu_2\} \\ \mathcal{D}_1 &= \{\nu_1, \nu_2, \nu_3\} \\ \mathcal{D}_2 &= \{\rho_1, \rho_2\} \\ (\_)^{M_S} &= \{(a, \mu_1), (f, \{(\mu_1, \mu_2)\})\} \\ (\_)^{M_1} &= \{(a, \nu_3), (b, \nu_1), (c, \nu_3), (f, \{(\nu_3, \nu_1), (\nu_1, \nu_2)\})\} \\ (\_)^{M_2} &= \{(a, \rho_2), (d, \rho_1), (e, \rho_2), (f, \{(\rho_2, \rho_1), (\rho_1, \rho_2)\})\} \end{aligned}$$

To unify M_1 with respect to M_S, we have $\mathcal{D}_1 \leftarrow \mathcal{D}_1[\mu_1/\nu_3]$ by applying the Const-Sub rule, and $\mathcal{D}_1 \leftarrow \mathcal{D}_1[\mu_2/\nu_1]$ by applying the Func-Sub rule. As the result, the updated M_1 is as follows.

$$\begin{aligned} \mathcal{D}_1 &= \{\mu_2, \nu_2, \mu_1\} \\ (\_)^{M_1} &= \{(a, \mu_1), (b, \mu_2), (c, \mu_1), (f, \{(\mu_1, \mu_2), (\mu_2, \nu_2)\})\} \end{aligned}$$

Analogously we have $M_2 \leftarrow$ `Unify(`M_2`,`M_S`)` such that:

$$\begin{aligned} \mathcal{D}_2 &= \{\mu_2, \mu_1\} \\ (\_)^{M_2} &= \{(a, \mu_1), (d, \mu_2), (e, \mu_1), (f, \{(\mu_1, \mu_2), (\mu_2, \mu_1)\})\} \end{aligned}$$

Algorithm 6: The procedure of `RefineSharedInterp`.

```
  Input   : $M_S, M_1, \ldots, M_k$
  Output  : The refined $M_S$
1 $F \leftarrow$ ExtractSFA($M_S, M_1, \ldots, M_k$);
2 foreach $(f, \mathbf{d}) \in F$ do
3     $f^{M_S} \leftarrow f^{M_S} \cup (\mathbf{d}, d_0)$ where $d_0 \in \mathcal{D}_S$;
4 return $M_S$
```

Table 2
The decision procedure by steps given the input problem ϕ and its lazy decomposition ψ.

No.	Operation	#Line	Input	Output
1	`Solve`	4	ψ_i	$s_1 = \text{sat}$; $s_2 = \text{sat}$; $s_3 = \text{sat}$
2	`ComputeSharedInterp`	8	G,ψ	$M_S = (\mathcal{D}_S, (\_)^{M_S})$ where $\mathcal{D}_S = \{\mu_1, \mu_2, \mu_3\}$, $(\_)^{M_S} = \{(a, \mu_1), (b, \mu_2), (c, \mu_3), (f, \{\})\}$
3	`Formulate`	10	M_S	$F_S : (a \neq b) \wedge (b \neq c) \wedge (a \neq c)$
4	`Interpolate`	13	(ψ_i, F_S) $1 \leq i \leq 3$	$(I_i, M_i, flag) = (-, (\mathcal{D}_i, (\_)^{M_i}), \textbf{false})$ where $\mathcal{D}_1 = \{\nu_1, \nu_2, \nu_3\}$, $(\_)^{M_1} = \{(a, \nu_1), (b, \nu_2), (c, \nu_3), (f, \{(\nu_2, \nu_3)\})\}$ $\mathcal{D}_2 = \{\rho_1, \rho_2, \rho_3\}$, $(\_)^{M_2} = \{(a, \rho_1), (b, \rho_2), (c, \rho_3), (f, \{(\rho_1, \rho_3), (\rho_3, \rho_1)\})\}$ $\mathcal{D}_3 = \{\eta_1, \eta_2, \eta_3, \eta_4, \eta_5\}$, $(\_)^{M_3} = \{(a, \eta_1), (b, \eta_2), (c, \eta_4), (f, \{(\eta_1, \eta_4), (\eta_2, \eta_5)\})\}$
5	`CombineInterp`	16	M_S,M_1,M_2,M_3	**false**
6	`RefineSharedInterp`	17	G,ψ	$\mathcal{D}_S = \{\mu_1, \mu_2, \mu_3, \mu_4, \mu_5\}$, $(\_)^{M_S} = \{(a, \mu_1), (b, \mu_2), (c, \mu_3), (f, \{(\mu_1, \mu_4), (\mu_1, \mu_5)\})\}$
7	`Formulate`	10	M_S	$F_S : \bigwedge_{t_1, t_2 \in \{a,b,c,f(a),f(b)\}} t_1 \neq t_2$
8	`Interpolate`	13	(ψ_i, F_S) $1 \leq i \leq 2$	$(I_1, M_1, flag) = (f(b) = c, -, \textbf{true})$ $(I_2, M_2, flag) = (f(a) = c, -, \textbf{true})$
9	update G	14	–	$G : (f(b) = c) \wedge (f(a) = c)$
10	`ComputeSharedInterp`	8	G,ψ	$\mathcal{D}_S = \{\mu_1, \mu_2, \mu_3\}$, $(\_)^{M_S} = \{(a, \mu_1), (b, \mu_2), (c, \mu_3), (f, \{(\mu_1, \mu_3), (\mu_2, \mu_3)\})\}$
11	`Formulate`	10	M_S	$F_S : (a \neq b) \wedge (b \neq c) \wedge (a \neq c) \wedge c = f(a) = f(b)$
12	`Interpolate`	13	(ψ_3, F_S)	$(I_3, M_3, flag) = (f(a) \neq f(b), -, \textbf{true})$
13	update G	14	–	$G : (f(b) = c) \wedge (f(a) = c) \wedge (f(a) \neq f(b))$
14	return	20	–	unsat

Next, we collect the shared function applications that have no valuations in M_S by calling `ExtractSFA(`M_1`,`M_2`,`M_S`)`. There are two shared function applications: (f, μ_1) and (f, μ_2). Since $f^{M_S}(\mu_1)$ is defined, the resultant set F consists of one element (f, μ_2). Therefore, the `CombineInterp` procedure returns **false**.

5.6. `RefineSharedInterp`: *refine the shared interpretation*

`RefineSharedInterp` refines the valuations of shared function symbols in M_S in order to make M_S have valuations on all the shared function applications over $M_1, \ldots, M_k$. Algorithm 6 illustrates its procedure which is to specify arbitrary values for shared function applications that formerly have no valuations in M_S. The refined model M_S' has an important property: $G^{M_S} = \text{true} \Longrightarrow G^{M_S'} = \text{true}$. This is because M_S' contains all the information in M_S which is sufficient to interpret G as true.

Example 5. Consider the formula and its decomposition in Example 2. First, the `ExtractSFA` procedure derives the set of shared function applications $F = \{(f, \mu_2)\}$. Next, we can refine the M_S by specifying a value for $f^{M_S}(\mu_2)$. The refined $M_S = (\mathcal{D}_S, (\_)^{M_S})$ is as follows.

$$\begin{aligned} \mathcal{D}_S &= \{\mu_1, \mu_2\} \\ (\_)^{M_S} &= \{(a, \mu_1), (f, \{(\mu_1, \mu_2), (\mu_2, \mu_1)\})\} \end{aligned}$$

And thus M_S can be characterized as the following formula:

$$F_S : a = f(f(a)) \wedge a \neq f(a)$$

5.7. *Running example*

We demonstrate the complete procedure of instantiated LDC using the following $\mathcal{T}_\mathsf{E}$-formula ϕ:

$$\underbrace{f(b) = c \wedge b \neq c}_{\psi_1} \wedge \underbrace{a = f(c) \wedge f(a) = c}_{\psi_2} \wedge \underbrace{a \neq b \wedge f(a) \neq f(b)}_{\psi_3}$$

Assume that the decomposition is $\psi = \{\psi_1, \psi_2, \psi_3\}$. Table 2 lists the detailed procedure of solving the satisfiability of ϕ. For the sake of clarity, we elaborate the step 6 as follows. After the `Unify` operation, M_1, M_2, M_3 are updated as

$$
\begin{aligned}
\mathcal{D}_1 &= \{\mu_1, \mu_2, \mu_3\} \\
\mathcal{D}_2 &= \{\mu_1, \mu_2, \mu_3\} \\
\mathcal{D}_3 &= \{\mu_1, \mu_2, \mu_3, \eta_4, \eta_5\} \\
(\_)^{M_1} &= \{(a, \mu_1), (b, \mu_2), (c, \mu_3), (f, \{(\mu_2, \mu_3)\})\} \\
(\_)^{M_2} &= \{(a, \mu_1), (b, \mu_2), (c, \mu_3), (f, \{(\mu_1, \mu_3), (\mu_3, \mu_1)\})\} \\
(\_)^{M_3} &= \{(a, \mu_1), (b, \mu_2), (c, \mu_3), (f, \{(\mu_1, \eta_4), (\mu_2, \eta_5)\})\}
\end{aligned}
$$

The `ExtractSFA` procedure returns the set $F = \{(f, \mu_1), (f, \mu_2)\}$ of shared function applications that have no valuations in M_S. The values for $f^{M_S}(\mu_1)$ and $f^{M_S}(\mu_2)$ are then specified and the shared interpretation M_S is refined as follows.

$$
\begin{aligned}
\mathcal{D}_S &= \{\mu_1, \mu_2, \mu_3, \mu_4, \mu_5\} \\
(\_)^{M_S} &= \{(a, \mu_1), (b, \mu_2), (c, \mu_3), (f, \{(\mu_1, \mu_4), (\mu_2, \mu_5)\}
\end{aligned}
$$

6. Discussion

In this section we analyze the LDC framework and its instantiation in $\mathcal{T}_{\mathsf{E}}$ from a theoretical perspective. First we will show that LDC is sound and complete, i.e. it always returns correct results. Then, we will prove that the instantiation in $\mathcal{T}_{\mathsf{E}}$ terminates after a finite number of steps. Finally, we discuss the generality of LDC.

6.1. Soundness and completeness

First, we begin with formal specifications of LDC interfaces.

`Decompose` is to make a k-decomposition of the formula ϕ, given ϕ and k:

$$\texttt{Decompose}(\phi, k) = \{\psi_1, \ldots, \psi_k\}$$

`Solve` is to determine the satisfiability of input formula ϕ:

$$\texttt{Solve}(\phi) = \begin{cases} \text{sat} & \phi \text{ is satisfiable} \\ \text{unsat} & \phi \text{ is unsatisfiable} \end{cases}$$

`ComputeSharedInterp` derives a shared interpretation M_S such that the domain of $(\_)^{M_S}$ is Σ_ψ and M_S satisfies the constraint G:

$$\texttt{ComputeSharedInterp}(G, \psi) = M_S \quad \begin{array}{l} \text{where } G^{M_S} = \text{true and} \\ \text{for each constant } c \in \Sigma_\psi, c^{M_S} \text{ is defined} \end{array}$$

`Formulate` is to convert the interpretation M_S into its *semantically equivalent formula* F_S. The concept of semantically equivalent formula is defined as follows.

Definition 4. Given an interpretation M and a formula ϕ, we say ϕ is *fully interpreted* by M iff each sub-term of ϕ can be interpreted as an element in the domain of M. We use the notation $\mathcal{L}_M$ to denote the set of all the formulae which can be fully interpreted by M.

Definition 5. Given an interpretation M, F is its semantically equivalent formula iff

$$\forall \phi \in \mathcal{L}_M . \phi^M = \text{true} \Longleftrightarrow (\phi \wedge F) \text{ is satisfiable}$$

We use the notation $M \rhd F$ to denote that the interpretation M and the formula F are semantically equivalent. Note that $\rhd$ is non-commutative.

Lemma 3. *Given an interpretation M and its semantically equivalent formula F, for a formula $\phi \in \mathcal{L}_M$ such that $\phi^M = \text{true}$, we have $F \to \phi$.*

Proof. By Definition 5, $(\neg\phi)^M = \text{false} \Longleftrightarrow (\neg\phi \wedge F)$ is unsatisfiable. Thus $(\neg F \vee \phi)$ is valid, and we have $F \to \phi$. □

Then, the formal specification of `Formulate` is as follows.

$$\texttt{Formulate}(M_S) = F_S \Longrightarrow M_S \rhd F_S$$

Furthermore, given an interpretation M, its semantically equivalent formula always exists. Consider two sets of formulae $\mathcal{F}_M = \{\phi \mid \phi^M = \text{true}\}$ and $\mathcal{F}_{\bar{M}} = \{\phi \mid \phi^M = \text{false}\}$, then we have $F = \left(\bigwedge_{\phi \in \mathcal{F}_M} \phi\right) \wedge \left(\bigwedge_{\rho \in \mathcal{F}_{\bar{M}}} \neg\rho\right)$ such that $M \rhd F$ holds.

`Interpolate` computes either an interpolant or an interpretation given two formulae.

$$\texttt{Interpolate}(F_1, F_2) = \begin{cases} (I, -, \textbf{true}) & F_1 \to I \to \neg F_2 \\ (-, M, \textbf{false}) & M \models (F_1 \wedge F_2) \end{cases}$$

`CombineInterp` is to check whether $M_1, \ldots, M_k$ are combinable with respect to the shared interpretation M_S where $M_S \rhd F_S$ and $M_i \models (\psi_i \wedge F_S)$ for each $1 \le i \le k$. It returns **true** if $M_1, \ldots, M_k$ are combinable, or **false** otherwise.

$$\texttt{CombineInterp}(M_S, M_1, \ldots, M_k) = \begin{cases} \textbf{true} & M_1, \ldots, M_k \text{ are combinable under } M_S \\ \textbf{false} & \text{Otherwise} \end{cases}$$

The *interpretation combinability* is formally defined as follows.

Definition 6 *(Interpretation combinability).* k interpretations $M_1, \ldots, M_k$ are combinable under the shared interpretation M_S if there exists $M_1' \cong M_1, \ldots, M_k' \cong M_k$ where $M_1' = (\mathcal{D}_1, (\_)^{M_1'}), \ldots, M_k' = (\mathcal{D}_k, (\_)^{M_k'})$ such that:

1. $\Sigma_{M_S} = \bigcup\limits_{1 \le i < j \le k} (\Sigma_{M_i} \cap \Sigma_{M_j})$;
2. $c^{M_i'} = c^{M_j'} = c^{M_S}$ for each constant $c \in \Sigma_{M_i} \cap \Sigma_{M_j}$ $(1 \le i < j \le k)$;
3. for each n-ary $(n > 0)$ function symbol $f \in \Sigma_{M_i} \cap \Sigma_{M_j}$ and $\mathbf{d} \in (\mathcal{D}_i \cap \mathcal{D}_j)^n$ $(1 \le i < j \le k)$, if $f^{M_i'}(\mathbf{d})$ and $f^{M_j'}(\mathbf{d})$ are defined, then $f^{M_S}(\mathbf{d})$ is also defined and we have $f^{M_i'}(\mathbf{d}) = f^{M_j'}(\mathbf{d}) = f^{M_S}(\mathbf{d})$.

Then, $M_1, \ldots, M_k$ can be combined into a new interpretation $M = (\mathcal{D}, (\_)^M)$ such that:

1. $\mathcal{D} = \bigcup\limits_{1 \le i \le k} \mathcal{D}_i$;
2. $c^M = c^{M_i'} = c^{M_j'} = c^{M_S}$ for each constant $c \in \Sigma_{M_i} \cap \Sigma_{M_j}$ $(1 \le i < j \le k)$;
3. $c^M = c^{M_i'}$ for each constant $c \in (\Sigma_{M_i} \setminus \Sigma_{M_S})$ $(1 \le i \le k)$;
4. $f^M = \bigcup\limits_{1 \le i \le s} f^{M_{l_i}'}$ for each n-ary $(n > 0)$ function symbol f such that $f \in \Sigma_{M_t}$ $(t = l_1, \ldots, l_s$ and $1 \le l_1 < \cdots < l_s \le k)$ and $f \notin \Sigma_{M_{t'}}$ $(1 \le t' \le k$ and $t' \notin \{l_1, \ldots, l_s\})$;
5. $f^M = f^{M_i'}$ for each n-ary $(n > 0)$ function symbol $f \in (\Sigma_{M_i} \setminus \Sigma_{M_S})$ $(1 \le i \le k)$.

The notation $M = M_1 \bigoplus\limits_{M_S} \cdots \bigoplus\limits_{M_S} M_k$ is used to denote that $M_1, \ldots, M_k$ are combined into a new interpretation M with respect to the shared interpretation M_S.

`RefineSharedInterp` is to compute a new shared interpretation by specifying values for shared function applications which are not specified by M_S. Let M_S' be the result of `RefineSharedInterp(`M_S,$M_1, \ldots, M_k$`)` and then we have:

1. $M_S' \neq M_S$;
2. $\Sigma_{M_S'} = \Sigma_{M_S}$;
3. for each constant $c \in \Sigma_{M_S}$, $c^{M_S} = c^{M_S'}$;
4. for each n-ary function $f \in \Sigma_{M_S}$, $f^{M_S} \subseteq f^{M_S'}$.

In what follows, we will prove the soundness and completeness of LDC framework, i.e. the LDC procedure returns correct result given an arbitrary input formula ϕ and the size of decomposition k, on the premise that the procedure terminates.

Lemma 4. *$\phi \to G$ holds during the LDC procedure.*

Proof. G is initialized as true on line 11 and trivially $\phi \to$ true holds. Since G is strengthened by conjoining interpolants which are logical consequences of ϕ, thus $\phi \to G$ holds. □

Theorem 5. *The LDC framework is sound and complete, on the premise that the decision procedure terminates.*

Proof. We need to prove the following statement:

$$\forall \phi, k \text{ such that } \texttt{LDC}(\phi, k) \text{ terminates}.\phi \text{ is satisfiable} \Longleftrightarrow \texttt{LDC}(\phi, k) = \mathsf{sat}$$

Then, given arbitrary formula ϕ and the size of decomposition k $(k > 0)$ such that `LDC(`ϕ, k`)` terminates, we wish to prove the following two statements:

$$\begin{array}{ll} \phi \text{ is satisfiable.} \implies \texttt{LDC}(\phi,k) = \text{sat} & \text{(Soundness)} \\ \texttt{LDC}(\phi,k) = \text{sat} \implies \phi \text{ is satisfiable} & \text{(Completeness)} \end{array}$$

where the soundness statement can be equivalently transformed as:

$$\texttt{LDC}(\phi,k) = \text{unsat} \implies \phi \text{ is unsatisfiable}$$

Suppose $\texttt{LDC}(\phi,k) = \text{unsat}$ holds. As Algorithm 1 shows, the procedure terminates on either line 5 or 20. In the former case, there exists a subformula ψ_i which is unsatisfiable. Then ϕ, as well as $\bigwedge_{j=1}^{k} \psi_j$, is also unsatisfiable. In the latter case, $G \to \bot$ holds. Since $\phi \to G$ holds by Lemma 4, we have $\phi \to \bot$ and thus ϕ is unsatisfiable.

Suppose $\texttt{LDC}(\phi,k) = \text{sat}$ holds, then the procedure terminates on either line 6 or 16. In the former case, each ψ_i is satisfiable and $\Sigma_\psi = \emptyset$. Let $M_i \models \psi_i$ for each i. Thus for two arbitrary interpretations M_i, M_j ($1 \leqslant i < j \leqslant k$), we can simply combine M_i and M_j since $\Sigma_{M_i} \cap \Sigma_{M_j} = \emptyset$. Thus, we can also combine $M_1, \ldots, M_k$ into an interpretation M such that

1. $M = (\mathcal{D}, (\_)^M)$ where $\mathcal{D} = \bigcup_{j=1}^{k} \mathcal{D}_j$, $\mathcal{D}_j$ is the domain of M_j;
2. for each (constant or function) symbol $s \in \left(\bigcup_{j=1}^{k} \Sigma_{M_j}\right)$, $s^M = s^{M_i}$ if $s \in \Sigma_{M_i}$.

Therefore ϕ is satisfiable. In the latter case, CombineInterp returns true, thus $M_1, \ldots, M_k$ are combinable under M_S. Thus we can also construct a total interpretation $M = M_1 \bigoplus_{M_S} \cdots \bigoplus_{M_S} M_k$ such that $\psi_i^M = \text{true}$ for each $1 \leqslant i \leqslant k$. Therefore, $\phi^M = \text{true}$ holds and ϕ is satisfiable. □

To establish the soundness and completeness of the instantiation of LDC in $\mathcal{T}_\mathsf{E}$, we need to prove (1) the soundness of 7 interfaces in $\mathcal{T}_\mathsf{E}$ (as their behaviors meet their specifications), and (2) LDC in $\mathcal{T}_\mathsf{E}$ terminates. In what follows, we focus on the former while the latter is discussed in Section 6.2. Decompose is sound as long as it produces k-decomposition, which is ensured by the decomposition algorithm. The soundness of Solve and Interpolate relies on the soundness of underlying SMT solver and interpolation engine, which is assumed. ComputeSharedInterp is also sound because (1) the underlying SMT solver is sound, and (2) M_S contains valuations for all the shared constants, which is ensured by Algorithm 3. The soundness for other interfaces, however, should be established carefully.

Lemma 6. *Formulate in $\mathcal{T}_\mathsf{E}$ is sound.*

We formalize the specifications for Unify and ExtractSFA before establishing the soundness of CombineInterp and RefineSharedInterp. First of all, we define equality of domain elements across two interpretations.

Definition 7. Given two interpretations M_1 and M_2 with domains $\mathcal{D}_1$ and $\mathcal{D}_2$, respectively. We say $\nu_1 \in \mathcal{D}_1$ equals to $\nu_2 \in \mathcal{D}_2$ across M_1 and M_2 (denoted by $\nu_1 \sim \nu_2$) if one of the following conditions holds:

1. $\nu_1 = \nu_2$;
2. there exists $c \in \Sigma_{M_1} \cap \Sigma_{M_2}$ such that $c^{M_1} = \nu_1$ and $c^{M_2} = \nu_2$;
3. there exists $f \in \Sigma_{M_1} \cap \Sigma_{M_2}$ where f is an n-ary ($n > 0$) function symbol, such that $f^{M_1}(\nu_{11}, \ldots, \nu_{1n}) = \nu_1$ and $f^{M_2}(\nu_{21}, \ldots, \nu_{2n}) = \nu_2$ where $\nu_{1i} \sim \nu_{2i}$ ($1 \leq i \leq n$).

For two vectors $\mathbf{d}_1 = (\nu_{11}, \ldots, \nu_{1n}) \in \mathcal{D}_1^n$ and $\mathbf{d}_2 = (\nu_{21}, \ldots, \nu_{2n}) \in \mathcal{D}_2^n$, we have $\mathbf{d}_1 \sim \mathbf{d}_2$ if and only if $\nu_{1i} \sim \nu_{2i}$ for each $1 \leq i \leq n$.

Unify is to convert an interpretation M into a new one M' with respect to the shared interpretation M_S, such that:

1. $M \cong M'$;
2. Let h be the isomorphism of M into M', $\mathcal{D}$ and $\mathcal{D}_S$ be the domains of M and M_S, respectively. For each $\nu \in \mathcal{D}$ and $\nu_S \in \mathcal{D}_S$, we have $h(\nu) = \nu_S \iff \nu \sim \nu_S$.

Lemma 7. *Unify is sound.*

Given $M_1, \ldots, M_k$ and the shared interpretation M_S, ExtractSFA returns a set F of shared function applications. All the function applications $(f, \mathbf{d})$ that meet the following requirements are in F.

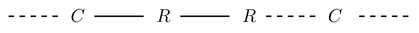


Fig. 3. A coarse-grained control-flow of LDC. Each node denotes an iteration of the main loop while C and R denote iterations calling `ComputeSharedInterp` and `RefineSharedInterp`, respectively.

1. f is an n-ary ($n > 0$) function symbol, $f \in \Sigma_{M_S}$ and $\mathbf{d} \in \mathcal{D}_S^n$;
2. there exists $1 \leq i < j \leq k$ such that:
   (a) $f \in \Sigma_{M_i} \cap \Sigma_{M_j}$;
   (b) there exists $\mathbf{d}_i \in \mathcal{D}_i^n$, $\mathbf{d}_j \in \mathcal{D}_j^n$ such that both $f^{M_i}(\mathbf{d}_i)$ and $f^{M_j}(\mathbf{d}_j)$ are defined, $\mathbf{d}_i \sim \mathbf{d}$, $\mathbf{d}_j \sim \mathbf{d}$ and $f^{M_S}(\mathbf{d})$ is not defined.

Lemma 8. `ExtractSFA` *is sound.*

Lemma 9. `CombineInterp` *in $\mathcal{T}_\mathsf{E}$ is sound.*

The instantiation of `RefineSharedInterp` is also sound. It is trivial that the conditions 2, 3, 4 of its specification hold. Assume that $M'_S = M_S$ holds, then `ExtractSFA` would return $\emptyset$ on line 1 in Algorithm 6, and thus the previous `CombineInterp` returns **true** and `RefineSharedInterp` would not be reached. Hence condition 1 also holds.

By Theorem 5, we have the following conclusion.

Lemma 10. *LDC in $\mathcal{T}_\mathsf{E}$ is sound and complete, on the premise that it terminates.*

6.2. Termination

In this subsection, we will prove that LDC in $\mathcal{T}_\mathsf{E}$ terminates. Algorithm 1 contains a main loop from line 10 to 21. In each iteration of the main loop, we compute a *new* shared interpretation M_S which helps to complete interpretations satisfying subformulae or interpolants which are logical consequences of the input formula. Informally, M_S marks the progress in decision procedure and thus the computation of M_S provides the core of the termination argument.

Iterations of the main loop can be abstracted into a coarse-grained control-flow sequence shown in Fig. 3. Each iteration either calls `ComputeSharedInterp` when there exists an inconsistent pair (ψ_i, F_S), or calls `RefineSharedInterp` when every (ψ_i, F_S) is consistent and M_S is required to be refined. We use C-node to denote the iteration calling `ComputeSharedInterp`, and R-node to denote the iteration calling `RefineSharedInterp`. Thus, the control-flow sequence consists of C-nodes and R-nodes while the number of nodes is the number of iterations for the main loop. In what follows, we demonstrate that the sequence has finite length for any input formula ϕ as the following statements hold:

1. between two neighboring C-nodes, there are a finite number of R-nodes;
2. there are a finite number of C-nodes in the sequence.

An *interpretation graph* can be constructed for the shared interpretation M_S. Without loss of generality, Σ_{M_S} consists of m constants $c_1, \ldots, c_m$ and n unary functions $f_1, \ldots, f_n$. The interpretation graph of M_S is a pair $G^{\mathsf{ig}} = (V^{\mathsf{ig}}, E^{\mathsf{ig}})$ of:

1. $V^{\mathsf{ig}} = \mathcal{D}_S$ where $\mathcal{D}_S$ is the domain of M_S;
2. $E^{\mathsf{ig}} = \{(v_1, v_2, i) \in V^{\mathsf{ig}} \times V^{\mathsf{ig}} \times \mathbb{N} \mid f_i^{M_S}(v_1) = v_2\}$. For $(v_1, v_2, i) \in E^{\mathsf{ig}}$, there is an edge from v_1 to v_2 with label i.

By the definition above, the out-degree of each node is no more than n. Moreover, the following lemmas show some important properties of G^{ig}.

Lemma 11. *Let $G^{\mathsf{ig}} = (V^{\mathsf{ig}}, E^{\mathsf{ig}})$ be the interpretation graph of shared interpretation M_S, and $V_C = \{v \mid c^{M_S} = v, c \in \{c_1, \ldots, c_m\}\}$. Then, for each $v \in (V^{\mathsf{ig}} \setminus V_C)$, there exists a path from one vertex in V_C to v.*

Proof. For each $v \in (V^{\mathsf{ig}} \setminus V_C)$, there exists $v' \in V^{\mathsf{ig}}$ and f_i $(1 \leqslant i \leqslant n)$ such that $f_i^{M_S}(v') = v$. Thus, there should be a subterm $f_i(t)$ of G where $(f_i(t))^{M_S} = v$ and $t^{M_S} = v'$. If t is a constant, then $\mathcal{P} = \langle v', v \rangle$ is the path. Otherwise, since t is finitely constructible, t has the form of $g_h(\ldots g_1(c) \ldots)$ where $g_1, \ldots, g_h \in \{f_1, \ldots, f_n\}$ and c is constant. If we have $c^{M_S} = v_1^*, g_1^{M_S}(v_1^*) = v_2^*, \ldots, g_h^{M_S}(v_h^*) = v'$ where $v_1^*, \ldots, v_h^* \in V^{\mathsf{ig}}$, then $\mathcal{P} = \langle v_1^*, \ldots, v_h^*, v', v \rangle$ is the path. $\square$

Lemma 12. *If $|V^{\mathsf{ig}}| = 1 + m \left(\sum_{i=0}^{N} n^i \right)$, there exists $v \in (V^{\mathsf{ig}} \setminus V_C)$, $v' \in V_C$ and the distance (i.e. length of the shortest path) from v' to v is no less than $(N + 1)$.*

Proof. We organize the vertices in V^{ig} in a stratified manner. First, we put all the vertices in V_C on the l_0 level. Then, all the vertices in V_1 are put on the l_1 level, where

$$V_1 = \{v \in V^{\mathrm{ig}} \mid \exists v' \in V_C.\text{the distance of } (v', v) \text{ is } 1\}$$

Since $|V_C| = m$, $|V_1| \leqslant n|V_C| = mn$. Analogously, we put all the vertices in V_N on the l_N level, where

$$V_N = \{v \in V^{\mathrm{ig}} \mid \exists v' \in V_C.\text{the distance of } (v', v) \text{ is } N\}$$

Since $|V_N| \leqslant n|V_{N-1}| \leqslant \cdots \leqslant n^N|V_C| = n^N m$, the number of vertices in $l_0, \ldots, l_N$ is no more than $m\left(\sum_{i=0}^{N} n^i\right)$. Thus, there exists at least one vertex on the l_j level where $j > N$. □

When `RefineSharedInterp` refines the shared interpretation M_S, shared function applications are specified with values, which introduces new edges in the interpretation graph. The length of the path from a vertex in V_C to a vertex in $V^{\mathrm{ig}} \setminus V_C$ is no more than $|S_{\psi_i}|$ because ψ contains no more than $|S_{\psi_i}|$ function application terms, each of which could extend the length of such path by 1. Let $N = \max\{|S_{\psi_i}| \mid 1 \leqslant i \leqslant k\}$, by Lemma 12 there should be no more than $m\left(\sum_{i=0}^{N} n^i\right)$ elements in $\mathcal{D}_S$. Then, we have the following theorem.

Theorem 13. *There are a finite number of R-nodes between two neighboring C-nodes.*

Proof. Since the distance for (v', v) is no more than N where $v' \in V_C$ and $v \in (V^{\mathrm{ig}} \setminus V_C)$, there are up to $m\left(\sum_{i=1}^{N} n^i\right)$ edges in the interpretation graph of M_S. Also, on an R-node, at least one new edge is introduced. Therefore the number of R-nodes between two neighboring C-nodes must be finite. □

On a C-node, there exists at least one inconsistent pair (ψ_i, F_S) from which we can derive an interpolant I_i such that $\psi_i \to I_i \to \neg F_S$. Hence, such inconsistency is prevented in the subsequent C-nodes because $I_i^{M_S} = \mathsf{false}$. To prove that the number of C-nodes is finite, we will prove that there are a finite number of shared interpretations.

If t is a term, we denote with $\mathsf{d}(t)$ the depth of function application, that is, $\mathsf{d}(t) = 0$ if t is a constant; $\mathsf{d}(t) = \mathsf{d}(t') + 1$ if t has the form $f(t')$. Let $S = \max\{\mathsf{d}(t) \mid t \text{ is subterm of } \phi\}$, we have the following lemma.

Lemma 14. *For each subterm t of G, $\mathsf{d}(t) \leqslant (N+1)S$ holds.*

Before proving this lemma, we introduce some concepts on interpolation. Let A and B be two formulae with respective signatures Σ_A and Σ_B, a symbol is *A-colored* if it is in $\Sigma_A \setminus \Sigma_B$, *$B$-colored* if it is in $\Sigma_B \setminus \Sigma_A$, and *transparent* if it is in $\Sigma_A \cap \Sigma_B$. A ground term t is: *A-colored* if it contains at least one A-colored symbol and others are transparent; *B-colored* if it contains at least one B-colored symbol and others are transparent; *AB-mixed* if it contains at least one A-colored symbol and one B-colored symbol; and *transparent* otherwise. Also, we use $\simeq$ to denote equality at the meta-level.

Proof of Theorem 14. Perform induction on C-nodes in the sequence. Let $P(F)$ amount that for each subterm t in the formula F, $\mathsf{d}(t) \leqslant (N+1)S$ holds.

Basis: On the first C-node, F_S consists of constants only. Thus for the interpolant I_i derived from the inconsistent pair (ψ_i, F_S), I_i also consists of constants. Since G is the conjunction of derived interpolants, $P(G)$ holds.

Induction step: By hypothesis, $P(G)$ holds on the previous C-node, then $P(F_S)$ also holds. If the previous R-nodes introduce new terms to F_S via `RefineSharedInterp`, we have $S \geqslant 1$ and for each new term t, $\mathsf{d}(t) \leqslant N$, thus $P(F_S)$ still holds. For the inconsistent pair (ψ_i, F_S) and its interpolant I_i, each subterm of I_i is (1) subterm of ψ_i or F_S; (2) a transparent term t such that $\psi_i \wedge F_S \vdash_{\mathcal{T}_{\mathsf{E}}} t_a \simeq t \wedge t \simeq t_b$ where t_a is a ψ_i-colored term, t_b is a F_S-colored term and $\psi_i \wedge F_S \vdash_{\mathcal{T}_{\mathsf{E}}} t_a \simeq t_b$ holds, by equality-interpolating property [31]. Interpolant could introduce new terms, which belongs to the latter case. Let T_N be the set of all new introduced terms from interpolants. $t_a \simeq t_b$ can be derived by (1) transitivity rule; (2) congruence rule, if $t_a \equiv f_i(t'_a)$, $t_b \equiv f_i(t'_b)$ and $\psi_i \wedge F_S \vdash_{\mathcal{T}_{\mathsf{E}}} t'_a \simeq t'_b$. If there exists a transparent term t' ($t' \in T_N$ or t' is a common subterm of ψ_i and F_S) such that $\psi_i \wedge F_S \vdash_{\mathcal{T}_{\mathsf{E}}} t'_a \simeq t' \wedge t' \simeq t'_b$, t can be $f_i(t')$, which may be a new term to be added into T_N. Since new introduced terms are transparent, t_a and t_b could only be the subterm of ψ_i and F_S respectively. For each (t_a, t_b), congruence rule is applied for no more than S times because $\mathsf{d}(t_a) \leqslant S$. The number of possible t_a is no more than N, thus congruence rule is applied for up to NS times to introduce auxiliary transparent terms. Since for each common subterm t_c of ψ_i and F_S we have $\mathsf{d}(t_c) \leqslant S$, the depths of new introduced terms are no more than $(N+1)S$. Hence, $P(G)$ still holds. □

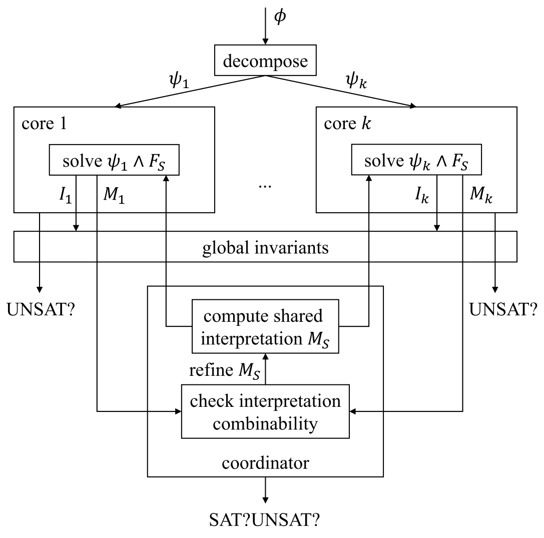


Fig. 4. The schematic overview of PZ3.

Therefore, on each C-node in the iteration sequence, the shared interpretation M_S derived by `ComputeSharedInterp` is the congruence relation on a subset of the term set $T = \{t \mid \mathsf{d}(t) \leqslant (N+1)S\}$. Since each $t \in T$ is constructed by shared symbols (no more than m constants and n functions), $|T|$ is bounded. Hence, the number of possible shared interpretations is finite.

Our conclusion can be generalized for the case where the arity of function symbol is more than 1. Therefore, we have the following theorems.

Theorem 15. *There are a finite number of C-nodes in the sequence.*

Theorem 16. *LDC in $\mathcal{T}_{\mathsf{E}}$ terminates within finite steps.*

Theorem 17. *LDC in $\mathcal{T}_{\mathsf{E}}$ is sound and complete.*

6.3. Generality

The LDC framework is irrelevant to a specific theory or solver. In principle, we can instantiate LDC in any theory upon any sequential solver. However, there are several limitations on the generality.

First, the underlying theory (or theory combination) should admit quantifier-free interpolation. In the theory of integers ($\mathcal{T}_{\mathbb{Z}}$), interpolant could introduce quantifiers. For example, suppose $\phi : y = 2x$ and $\psi : y = 2z + 1$, there does not exist an interpolant without quantifiers. Quantifier-free fragments of some theories, such as $\mathcal{T}_{\mathbb{Z}}$ and the theory of array ($\mathcal{T}_{\mathsf{A}}$) [22], are not closed under Craig interpolation. By the fact that quantifier elimination implies quantifier-free interpolation [32], theories such as $\mathcal{T}_{\mathsf{E}}$, theory of reals ($\mathcal{T}_{\mathbb{R}}$), theory of rationals ($\mathcal{T}_{\mathbb{Q}}$), theory of integer difference logic ($\mathcal{T}_{\mathsf{IDL}}$) and theory of recursive data structure ($\mathcal{T}_{\mathsf{RDS}}$) admit quantifier-free interpolation. Bruttomesso et al. also show that strong amalgamability guarantees the modularity of quantifier-free interpolation [32], which guides the construction of theory combination admitting quantifier-free interpolation.

Second, it is difficult to guarantee that the instantiation terminates given a formula of finite size in some underlying theories.

7. Experimental evaluation

The LDC framework is implemented as a parallel SMT solver PZ3, which is based on a widely used open-source SMT solver Z3 [13]. The following aspects of PZ3 are studied: (1) speed-up over Z3 on various problems; (2) parallel efficiency and its influence factors, especially the sparseness of input formulae; (3) comparison between LDC and the state-of-the-art portfolio approach [33].

7.1. Solver implementation

PZ3 is implemented upon APIs provided by Z3 4.3.3, as Z3 exposes various functionalities such as satisfiability testing, interpolation and formula simplification via APIs. For now PZ3 only supports quantifier-free equality logic with uninterpreted

Table 3
The list of experiments designed for PZ3. **Q** lists the questions to be answered by each experiment. **Dataset** shows the benchmarks on which the certain experiment is based. **Sect.** reports the subsection where the details of the certain experiment are demonstrated. **#Core** enumerates the number(s) of cores specified for the parallel solver.

Task	**Q**	**Dataset**	**Sect.**	**#Core**
compare the performances of PZ3 and Z3	1, 2	QF_UF in SMT-LIB	7.3	2–12
evaluate the parallel efficiency of PZ3 over Z3	2	random problems	7.4	4
compare the performances of PZ3, PZ3oc and Z3	3	QF_UF in SMT-LIB	7.5	2, 4, 8
time profiling for PZ3	4	QF_UF in SMT-LIB	7.6	2, 4, 8
compare the speed-up ratios of PZ3 and PCVC4 over their sequential solvers	5	QF_UF in SMT-LIB	7.7	4

functions. The parallelization is implemented using the POSIX threads library. The schematic overview of PZ3 is shown in Fig. 4. The input formula ϕ is decomposed into k subformulae, then k cores are used if possible while each of them runs a Z3 instance assigned with a subformula.[2] Moreover, there is a coordinator thread that performs shared symbol reasoning, including derivation of shared interpretation and checking for interpretation combinability. The coordinator blocks all other threads when it is working, thus it can either work on a standalone core (if available) or core i $(1 \leq i \leq k)$.

The source code, documents and evaluation data are all publicly available at `http://git.io/hpZg`.

7.2. Experimental setup

The evaluation is conducted on a server under Ubuntu 16.04, using Intel(R) Xeon(R) CPU E5-2603v3@1.60GHz (with 12 cores) and 96GB memory. We assess PZ3 by answering the following research questions:

1. Can PZ3 outperform its baseline Z3?
2. How does the parallel efficiency vary with (a) the number of cores, (b) the problem structure?
3. Where does the speed-up of PZ3 come from?
4. How are PZ3's time costs distributed to working phases and operations?
5. How effective is LDC compared to other parallelization approaches?

To answer the questions above, 5 experiments are designed as listed in Table 3. The first experiment compares the performances of PZ3 and its baseline Z3 using different numbers of cores to evaluate PZ3's speed-up over Z3 and its scalability. The second one is to measure PZ3's parallel efficiency on a set of randomly generated problems in order to validate our hypothesis on the relationship between problem structure and PZ3's parallel efficiency. In the third experiment, we assess how (1) parallelism and (2) algorithmic effect (i.e. workload reduction by lazy decomposition and conciliation) contribute to PZ3's speed-up by running a special version of PZ3 (namely PZ3oc) with all its threads specified to a single core, along with the normal PZ3 and Z3. The fourth experiment runs time profiling for PZ3. In the final experiment, we choose PCVC4, a parallel solver of CVC4 [15] as the representative of portfolio-based solver to compare LDC with, as the portfolio approach [33] is the state-of-the-art.

All the experiments except the second one are based on the `QF_UF` benchmarks of SMT-LIB.[3] The benchmark problems are originated from three application domains:

1. **minimum transitivity constraints (MTC):** 100 hierarchical problems containing contradictory cycles [34];
2. **automatic theorem proving (ATP):** problems obtained by trying to find a finite model of first-order formulae. There are 3 subcategories: SEQ, PEQ and NEQ with respective 56, 47, 48 problems.
3. **quasigroup (QG):** problems in loop theory and quasigroup theory [35]. There are 4 subcategories: loops6, QG5, QG6 and QG7 with respective 448, 5286, 244, 418 problems.

Details on the randomly generated problems for the second experiment are demonstrated in Section 7.4. In experiments, the timeout for each problem is set as 600 s by default.

7.3. Evaluation of parallel efficiency

Basic concepts. The parallel efficiency of PZ3 is defined as:

$$\eta = \frac{T_{\text{seq}}}{T_{\text{par}} \cdot n}$$

[2] In the rest of paper, the statement that PZ3 works using k cores implies that the input formula is decomposed into k subformulae if possible.
[3] `http://smt-lib.org`.

Table 4
The average time costs of Z3 (as the column "1" shows) and PZ3 using k cores ($2 \le k \le 12$, as the columns "2",…,"12" show) in various categories of the QF_UF benchmarks.

category	Σ	average time (s)											
		1	2	3	4	5	6	7	8	9	10	11	12
MTC	100	478.79	484.92	484.58	205.28	112.29	37.49	9.07	5.81	5.13	5.84	6.88	8.21
NEQ	48	47.36	46.94	51.82	64.91	47.91	55.32	68.18	66.30	50.64	49.75	72.38	75.83
PEQ	47	174.10	176.73	189.49	195.67	207.11	212.45	240.30	229.21	207.61	198.57	201.47	192.42
SEQ	56	29.79	32.62	55.60	56.32	79.01	84.92	95.25	115.77	103.91	79.60	76.92	92.25
loops6	448	0.26	0.53	0.67	0.55	0.55	0.60	0.61	0.70	1.04	1.10	1.40	1.74
QG5	5286	0.93	0.65	0.79	0.66	0.68	0.78	0.91	0.86	0.73	0.80	0.86	0.88
QG6	244	8.88	6.30	8.39	5.66	5.53	5.92	6.43	5.58	5.81	6.06	6.36	7.80
QG7	418	9.67	5.67	8.70	8.88	9.12	8.76	8.75	9.89	9.35	8.97	10.27	11.01

Table 5
The numbers of the solved problems by Z3 (as the column "1" shows) and PZ3 using k cores ($2 \le k \le 12$, as the columns "2",…,"12" show) in various categories of the QF_UF benchmarks.

category	Σ	#solved instance											
		1	2	3	4	5	6	7	8	9	10	11	12
MTC	100	21	20	20	72	88	100	100	100	100	100	100	100
NEQ	48	47	47	45	45	45	45	43	45	45	45	45	43
PEQ	47	37	36	35	33	33	31	30	31	32	33	33	33
SEQ	56	55	55	52	51	50	50	49	48	49	49	49	48
loops6	448	448	448	448	448	448	448	448	448	448	448	448	448
QG5	5286	5286	5286	5286	5286	5286	5286	5286	5286	5286	5286	5286	5286
QG6	244	244	244	244	244	244	244	244	244	244	244	244	244
QG7	418	418	418	418	418	418	418	418	418	418	418	418	418

where n is the number of cores in use, T_{seq} and T_{par} refer to the solving time of Z3 and PZ3, respectively. The perfect parallelization has $\eta = 1$. If $\eta < \frac{1}{n}$, Z3 outperforms PZ3. The speed-up is super-linear if $\eta > 1$. Thus, PZ3 has good scalability if η keeps to be near 1.0 as the number of cores increases. The speed-up ratio of PZ3 is defined as $\frac{T_{\text{seq}}}{T_{\text{par}}}$ which does not take the number of cores into consideration.

On the set of problems $\mathcal{X}$, the parallel efficiency of PZ3 is computed by:

$$\eta = \frac{\sum_{\phi \in \mathcal{X}'} T_{\text{seq}}^{\phi}}{n \sum_{\phi \in \mathcal{X}'} T_{\text{par}}^{\phi}}$$

where $\mathcal{X}' \subseteq \mathcal{X}$ excludes problems on which both solvers run out of time (because it is indeterminable which solver is advantageous). T_{seq}^{ϕ} and T_{par}^{ϕ} denote the time costs of Z3 and PZ3 on the problem ϕ, respectively.

Experimental design. We run PZ3 and Z3 on the QF_UF benchmarks while PZ3 is configured to use 2–12 cores. To answer Question 1, we compare the time costs of PZ3 and Z3 on the benchmark problems. To answer Question 2 (a), we analyze how PZ3's parallel efficiency changes as the number of cores in use increases.

Results and discussion. Table 4 and Table 5 show the comparison results on the solving time and the number of solved problems, respectively. Furthermore, we use scatter plots in logarithmic scale shown in Fig. 5 to illustrate the detailed comparison results in three problem categories.

***MTC*:** PZ3 shows significant superiority in this category using k ($k \ge 4$) cores. Z3 can only solve 21 problems while PZ3 can solve all 100 problems when $k \ge 6$. In particular, when $k = 9$, PZ3 uses only 5.13 s on each problem averagely. When $k = 2, 3$, PZ3 fails to outperform Z3 because subformulae are still difficult to be solved. Fig. 5 also shows that for the problems solved by Z3 within 1 s, PZ3 generally takes more time. This is because they have low problem complexities and thus PZ3 can hardly benefit from decomposition on them.

***ATP*:** In general, PZ3 is disadvantageous in this category as Z3 outperforms PZ3 on both the solving time and the number of solved problems. However, Fig. 5 shows that PZ3 has better performance than Z3 on a large portion of problems which are difficult for Z3 (more specifically, with more than 10^4s of time cost).

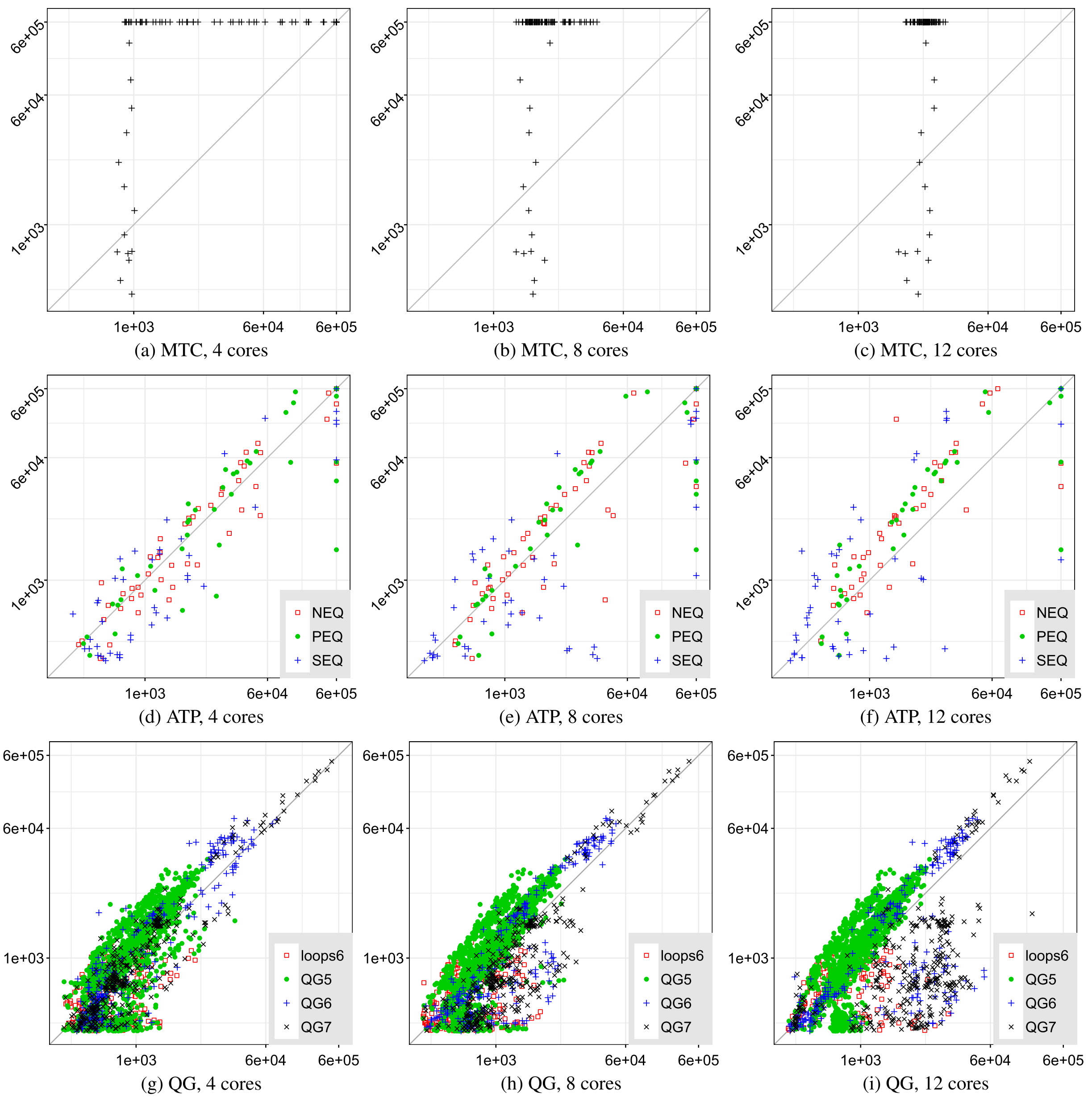


Fig. 5. Comparison of PZ3 and Z3 on the QF_UF benchmarks in terms of runtime. For each scatter plot, the x-axis and y-axis denote the time costs (in milliseconds) of PZ3 and Z3, respectively. The caption of each scatter plot reports the problem category and the number of cores in use.

Table 6
The ratio of problems on which PZ3 achieves speed-up, with respect to the number of cores.

#core	2	3	4	5	6	7	8	9	10	11	12
ratio	0.567	0.383	0.532	0.507	0.397	0.393	0.376	0.459	0.392	0.394	0.388

QG: Table 5 shows that both PZ3 and Z3 are capable to solve all the problems in this category. On the one hand, PZ3 is disadvantageous in the loops6 subcategory because the majority of its problems (332 out of 448) can be solved by Z3 within 300 ms while PZ3 has inherent overhead on decomposition and thread manipulation. On the other hand, PZ3 is generally more efficient in the QG5-QG7 subcategories. Furthermore, Fig. 5 also shows that PZ3 has an advantage on the problems difficult for Z3 (requiring more than 10^4s).

Table 6 lists the ratio of problems on which PZ3 has improved performance, with respect to the number of cores. Problems solved within 300 ms by both PZ3 and Z3 are not included because their differences can be substantially influenced

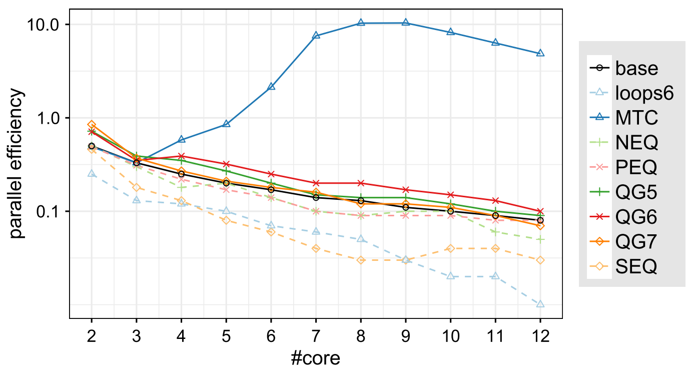


Fig. 6. The scalability of PZ3 in various categories of the `QF_UF` benchmarks.

by random errors in timing. The results show that PZ3 succeeds in achieving speed-up on more than one half of problems when the number of cores in use is 2, 4 or 5. Under other core configurations, the ratio ranges from 0.376 to 0.459.

Fig. 6 shows the parallel efficiency of PZ3 across 2–12 cores. The black base line illustrates the parallel efficiency $\frac{1}{n}$ denoting that the time costs of PZ3 and Z3 are the same with respect to the number of cores n. Overall, PZ3 has super-linear speed-up in the MTC category and generally outperforms Z3 in subcategories QG5-7 (shown in solid lines). However, PZ3 is disadvantageous in other subcategories (shown in dashed lines). In general, the parallel efficiency decreases as n grows when n is sufficiently large. This can be explained as follows. First, larger n makes shared symbol reasoning more time-expensive in checking interpretation combinability and refining shared interpretation while shared symbol reasoning contributes to the unparallelized part of PZ3's execution. Second, larger n leads to a larger portion of shared symbols, thus more conciliation iterations are generally required. However, we also notice that, when $4 \leq n \leq 8$, the parallel efficiency increases as n grows in the MTC category. This is because the difficulties of subproblems are substantially reduced compared to the original problem as the decomposition becomes more fine-grained.

Summary. For Question 1, PZ3 generally outperforms Z3 in 4 out of 8 subcategories of the `QF_UF` benchmarks. In particular, PZ3 is orders of magnitude faster than Z3 in the MTC category. For Question 2 (a), the parallel efficiency of PZ3 typically decreases as the number of cores grows. However, for problems with sparse structures such as problems in the MTC category, PZ3 can benefit from the extra cores when the number of cores is small.

Admittedly, PZ3 cannot outperform Z3 on all kinds of problems. However, in practice, one can always build a portfolio solver with one PZ3 instance and one Z3 instance. Two solver instances work on the input problem independently and the portfolio solver terminates once any solver instance terminates. Thus, the portfolio solver can take the advantage of PZ3 while preserving Z3's performance in the worst case. Furthermore, one can also measure the sparseness of the input formula (Section 7.4) to choose the appropriate solver favorable to the certain problem.

7.4. Speed-up and formula sparseness

Basic concepts. We have an important observation from the experimental results in the previous subsection: PZ3 tends to have an advantage on problems with sparse structure (e.g. problems in the MTC category). Intuitively, a formula is sparse if, for its two arbitrary clauses, few symbols are shared. First of all, the term "sparse" needs to be defined formally.

Definition 8 *(Equality graph).* Let ϕ be a $\mathcal{T}_{\mathsf{E}}$-formula without uninterpreted functions and $\mathcal{E}_\phi$ be ϕ's literal set consisting of equalities and inequalities. The undirected graph $G^{\mathsf{E}}_\phi = (V^{\mathsf{E}}_\phi, E^{\mathsf{E}}_\phi)$ is the *equality graph* of ϕ where V^{E}_ϕ is the set of constants in ϕ and $E^{\mathsf{E}}_\phi = \{(v_i, v_j) \mid v_i, v_j \in V^{\mathsf{E}}_\phi, (v_i = v_j), (v_j = v_i), (v_i \neq v_j)$ or $(v_j \neq v_i)$ is in $\mathcal{E}_\phi\}$.

Note that the concept of equality graph here is different from the concept with the same name in [34] because ϕ is not required to be in NNF and we do not distinguish equality and inequality as two kinds of edges. If ϕ contains uninterpreted functions, we use Ackermann's reduction [36] to convert ϕ to an equi-satisfiable $\mathcal{T}_{\mathsf{E}}$-formula ϕ' with uninterpreted functions reduced, then we have $G^{\mathsf{E}}_\phi = G^{\mathsf{E}}_{\phi'}$.

Definition 9 *(Constant sparseness).* Let ϕ be a $\mathcal{T}_{\mathsf{E}}$-formula in CNF: $\phi_1 \wedge \cdots \wedge \phi_k$. ϕ contains no uninterpreted functions and is simplified such that a clause does not contain multiple duplicated literals, or a literal and its negation simultaneously. The constant sparseness γ_ϕ of ϕ is defined as follows:

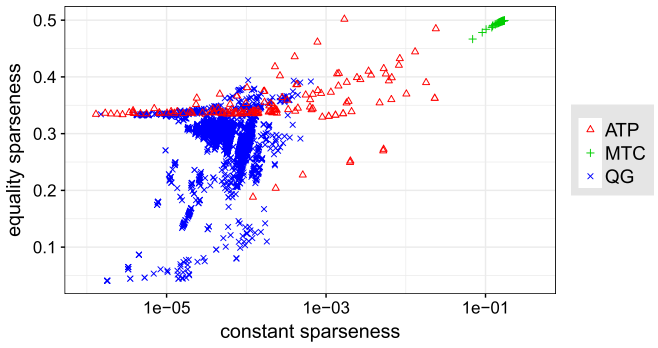


Fig. 7. The sparseness distribution of the `QF_UF` benchmarks from SMT-LIB.

$$\gamma_\phi = \frac{|V^{\mathsf{E}}_\phi| - V_{\min}}{V_{\max} - V_{\min}}$$

where $V_{\max} = 2 \sum_{1 \le i \le k} |\mathcal{E}_{\phi_i}|$ and $V_{\min} = \left\lceil \frac{1 + \sqrt{1 + 8|E^{\mathsf{E}}_\phi|}}{2} \right\rceil$.

Since ϕ contains $N = \sum_{1 \le i \le k} |\mathcal{E}_{\phi_i}|$ literals in total, then ϕ could contain at most $2N$ constants. Let M be the number of constants in G^{E}_ϕ, then we have $\frac{M(M-1)}{2} \ge |E^{\mathsf{E}}_\phi|$ where $\frac{M(M-1)}{2}$ is the largest number of different equalities over M constants. Thus $V_{\min}$ is the minimum number of constants possible for G^{E}_ϕ.

We have $\gamma_\phi \in [0, 1]$. When γ_ϕ is near 0, clauses share a large portion of symbols in Σ_ϕ and thus PZ3 is probably disadvantageous because the search space of shared interpretations can be hardly reduced compared to the interpretation space of the original problem. When γ_ϕ is near 1, it is easier for PZ3 to find a decomposition with few shared symbols and thus the conciliation step can terminate fast.

Definition 10 *(Equality sparseness).* Let ϕ be a $\mathcal{T}_{\mathsf{E}}$-formula in CNF: $\phi_1 \wedge \cdots \wedge \phi_k$. ϕ contains no uninterpreted functions and is simplified. The equality sparseness δ_ϕ of ϕ is defined as follows:

$$\delta_\phi = \frac{|E^{\mathsf{E}}_\phi| - E_{\min}}{E_{\max} - E_{\min}}$$

where $E_{\max} = \sum_{1 \le i \le k} |\mathcal{E}_{\phi_i}|$ and $E_{\min} = \max_{1 \le i \le k} |\mathcal{E}_{\phi_i}|$.

Also, we have $\delta_\phi \in [0, 1]$. When δ_ϕ is near 0, ϕ can probably be more efficiently solved by Z3 because it is easier for a lazy solver to perform theory propagation and clause learning by deciding truth values of some literals. When δ_ϕ is near 1, however, the solver has to traverse the whole interpretation space while theory propagation and clause learning are not effective.

Fig. 7 is a scatter plot showing the distribution of the problems in the `QF_UF` benchmarks for the two sparseness metrics. The x-axis refers to constant sparseness and it is in logarithmic scale. The y-axis refers to equality sparseness and it is linear.

***MTC*:** Problems have large constant and equality sparseness. PZ3 can efficiently solve the most of them with decompositions with few shared symbols. Moreover, Z3 is inefficient on MTC problems because they are unsatisfiable problems with large equality sparseness, thus Z3 has to traverse exponential number of candidate interpretations while theory propagation and clause learning are not effective.

***ATP and QG*:** Problems have generally low constant sparseness. This is because Ackermann's reduction introduces auxiliary constants along with additional constraints of function congruence for a large number of applications of uninterpreted functions, and thus the equality graph becomes dense. A possible explanation for the fact that PZ3 cannot substantially outperform Z3 on these problems is that the decomposition contains a large portion of shared symbols, thus more conciliation iterations are generally required to find a witness of global satisfiability or the inconsistency over the shared symbols. It is noteworthy that there are many factors that can influence PZ3's performance in addition to the sparseness, thus a large portion of QG problems can still be favorable to PZ3 (and we will give an explanation in Section 7.5).

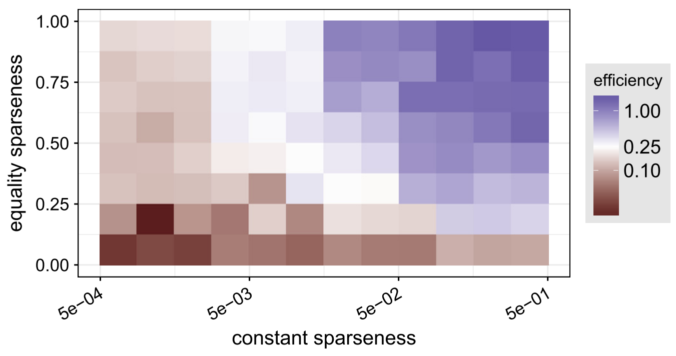


Fig. 8. The distribution of parallel efficiency over two sparseness metrics on the randomly generated problems. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Experimental design. Our hypothesis is that for a $\mathcal{T}_{\mathsf{E}}$-formula ϕ, PZ3 has more chance to outperform Z3 as γ_ϕ and δ_ϕ increase. To validate this hypothesis, we compare the performances of PZ3 (using 4 cores) and Z3 on a set of randomly generated problems with different sparseness values while other factors that could influence the solver's performance are carefully controlled. A generated problem is in CNF with N_v constants, no uninterpreted functions and N_c clauses, each clause consists of N_l different literals. Problem generation is parametric as N_c, N_l, γ_ϕ and δ_ϕ need to be specified. In our experiment we have $N_c = 3000$ and $N_l = 3$. $\gamma_\phi = 5 \times 10^{-i}$ where i is uniformly distributed over $[1, 4]$ and δ_ϕ is uniformly distributed over $[0, 1]$. The problem generator constructs an equality graph G^{E}_ϕ by the given N_c, N_l, γ_ϕ and δ_ϕ, then distributes (in)equalities represented by edges in E^{E}_ϕ to the clauses. We generate totally 3600 random problems for evaluation.

Results and discussion. Fig. 8 is a heat map visualizing PZ3's parallel efficiency with respect to two sparseness metrics. For better visualization, we choose 96 cluster centers and then assign each problem to the nearest cluster center, next we compute the parallel efficiency for problems assigned to each cluster center. The coordinate of each cluster center is

$$\left(5 \times 10^{\left(\frac{2k+1}{8}-4\right)}, \frac{2l+1}{16}\right) \text{ where } k = 0 \ldots 11 \text{ and } l = 0 \ldots 7$$

In Fig. 8, a block is used to represent the problems assigned to the certain cluster center while its color represents PZ3's parallel efficiency on these problems. As the constant sparseness increases, PZ3 generally has better parallel efficiency because PZ3 can take advantage of decompositions with few shared constants. The color of block is closer to red as the equality sparseness decreases. This is mainly because the problems with smaller equality sparseness are more favorable to Z3 by taking advantage of theory propagation and clause learning. Moreover, it can be observed that PZ3 can achieve super-linear speed-up on problems with large equality and constant sparseness. All in all, the visualization results substantiate our hypothesis.

Summary. Given a $\mathcal{T}_{\mathsf{E}}$-formula ϕ, PZ3 has more chance to solve it faster than Z3 when ϕ has larger constant and equality sparseness.

Sparse problems are of practical concern. The majority of real-world software/hardware systems are organized in a modular manner. Modules are loosely-coupled as they interact with each other by sharing a few variables/signals. Thus a verification task on a real-world system, which is not feasible for existing sequential SMT solvers, is favorable to LDC because the inter-module logic can be handled efficiently via conciliation.

7.5. Speed-up factor analysis

Basic concepts. The speed-up of PZ3 has two main sources: (1) parallelism and (2) the algorithmic effect of LDC. To quantify the contributions of these two sources, we run (1) Z3, (2) PZ3 with all the threads assigned to separated cores, and (3) PZ3oc with all the threads assigned to a single core on the same benchmarks and compare their performances. Let T_1, T_2, T_3 be the time costs of Z3, PZ3 and PZ3oc on the problem ϕ, respectively. Without regarding inherent overhead in thread manipulation, we have

$$T_3 = \lambda \cdot T_1$$

where λ is the *workload ratio* on ϕ. If $\lambda > 1$, solving ϕ by LDC has more workload than solving ϕ directly, and vice versa otherwise. We also have

Table 7
The comparison results of Z3, PZ3 and PZ3oc on the `QF_UF` benchmarks. WR and CPE refer to workload ratio and canonical parallel efficiency, respectively.

#core	MTC					ATP					QG				
	average time (s)			WR	CPE	average time (s)			WR	CPE	average time (s)			WR	CPE
	Z3	PZ3	PZ3oc			Z3	PZ3	PZ3oc			Z3	PZ3	PZ3oc		
2	8.468	24.596	30.152	3.561	0.613	35.486	33.232	42.308	1.192	0.637	1.762	1.186	1.485	0.843	0.626
4	22.790	0.733	0.841	0.037	0.287	25.278	17.705	30.978	1.225	0.437	1.771	1.375	2.119	1.196	0.385
8	22.790	3.008	3.165	0.139	0.132	31.651	30.561	50.622	1.599	0.207	1.787	1.558	2.712	1.518	0.218

decomposition	subproblem solving			shared symbol reasoning		
`Decompose`	`Solve`	`Interpolate`	`Formulate`	`CSI`	`CI`	`RSI`
decomposition	satisfiability checking	interpolant extraction	formulation	shared symbol reasoning		

Fig. 9. Bases for PZ3's time profiling with respect to working phases, LDC interfaces and basic operations from rows 1–3. `CSI`, `CI` and `RSI` are the abbreviations of `ComputeSharedInterp`, `CombineInterp` and `RefineSharedInterp`, respectively. Alignments of columns across the rows visualize the relationship of the profiling bases.

$$\eta_c = \frac{T_3}{T_2 \cdot n}$$

where n is the number of cores in use and η_c denotes *canonical parallel efficiency*, which is different from the previously defined parallel efficiency η as two time values to be compared are based on the same workload. Then, we have

$$\eta = \frac{T_1}{T_2 \cdot n} = \frac{\eta_c}{\lambda}$$

where η_c and λ measure the contributions of parallelism and the algorithmic effect, respectively. We say parallelism contributes more to the speed-up of PZ3 if $\frac{1}{\eta_c} < \lambda$ holds, while algorithmic effect contributes more if $\frac{1}{\eta_c} > \lambda$.

Experimental design. We compare the performances of Z3, PZ3 and PZ3oc on the `QF_UF` benchmarks using 2, 4, 8 cores. When n cores are in use, the timeout for PZ3oc is set to 600n seconds. To precisely approximate workload of LDC-based solving, only problems on which none of three solvers run out of time are included in speed-up factor analysis.

Results and discussion. Table 7 summarizes the comparison results in different problem categories.

***MTC*:** PZ3 has a significant speed-up over Z3 when the number of cores is 4 or 8, and algorithmic effect is the dominant factor contributing to the speed-up. This is because (1) subproblems are much easier to be solved than the whole problem, and (2) few symbols are shared over subproblems, thus the conciliation step is generally efficient.

***ATP*:** PZ3 outperforms Z3 on every core configuration, and parallelism contributes more to the speed-up as LDC has more workload than direct SMT solving.

***QG*:** Algorithmic effect of LDC has more influence on the speed-up when using 2 cores because for a large portion of problems, the unsatisfiability is derived directly from the unsatisfiability of a subproblem (see Table 8 for details). When the number of cores is 4 or 8, however, the speed-up is mainly contributed to by parallelism.

Table 7 also shows that in each category, the canonical parallel efficiency decreases as the number of cores grows, which implies that more time is wasted by idle threads. There are two main sources of thread idleness. First, workloads for subproblems are probably unbalanced since our decomposition is an approximate solution that minimizes the number of shared symbols, while balancing of subproblem complexity is not considered. Second, the coordinator contributes to the unparallelized part of PZ3's execution, as it blocks all other threads when performing shared symbol reasoning.

Summary. In the MTC category, algorithmic effect plays the dominant role in the speed-up of PZ3 as the decomposition substantially reduces the complexity of input problem. In the QG category, algorithmic effect contributes more to PZ3's speed-up because of the unsatisfiability of subproblems. In other cases, parallelism has more influences on PZ3's speed-up.

7.6. Profiling analysis

Basic concepts. There are several bases for profiling PZ3, as Fig. 9 shows. Since the LDC algorithm (Algorithm 1) can be roughly divided into three major working phases: decomposition (line 2), subproblem solving (lines 3–6, 10–14) and shared symbol reasoning (lines 7–8, 15–21), PZ3's wall time can be profiled with respect to these phases. The last 2

Table 8
The average wall time percentage of each working phase when 2/4/8 cores are in use on the `QF_UF` benchmarks. DC, SUB and SSR refer to decomposition, subproblem solving and shared symbol reasoning, respectively. #NC refers to the number of problems without conciliation. For each core number, the first (second) row reports the average wall time percentage with NC problems included (excluded).

#core	MTC (%)				ATP (%)				QG (%)			
	DC	SUB	SSR	#NC	DC	SUB	SSR	#NC	DC	SUB	SSR	#NC
2	1.05	91.64	7.32	1	2.81	96.96	2.32	126	15.83	81.67	2.49	3783
	0.78	91.51	7.7		3.47	93.86	2.66		17.06	76.83	6.11	
4	0.53	96.35	3.12	1	4.53	92.98	2.49	84	15.09	82.21	2.69	3622
	0.49	96.35	3.16		3.45	89.3	7.24		13.45	80.23	6.31	
8	0.89	95.14	3.97	1	9.09	88.85	2.06	86	17.31	79.15	3.54	2605
	0.77	95.21	4.01		4.28	87.75	7.98		14.82	79.02	6.16	

Table 9
The average percentage of actual CPU time of each LDC interface and basic operation when 2/4/8 cores are in use on the `QF_UF` benchmarks. DC, FM, SC, IE, SSR refer to decomposition, formulation, satisfiability checking, interpolant extraction and shared symbol reasoning, respectively. `Slv` and `Itp` refer to `Solve` and `Interpolate`, respectively. For each core number, the first (second) row reports the average percentage of actual CPU time with the problems without conciliation included (excluded).

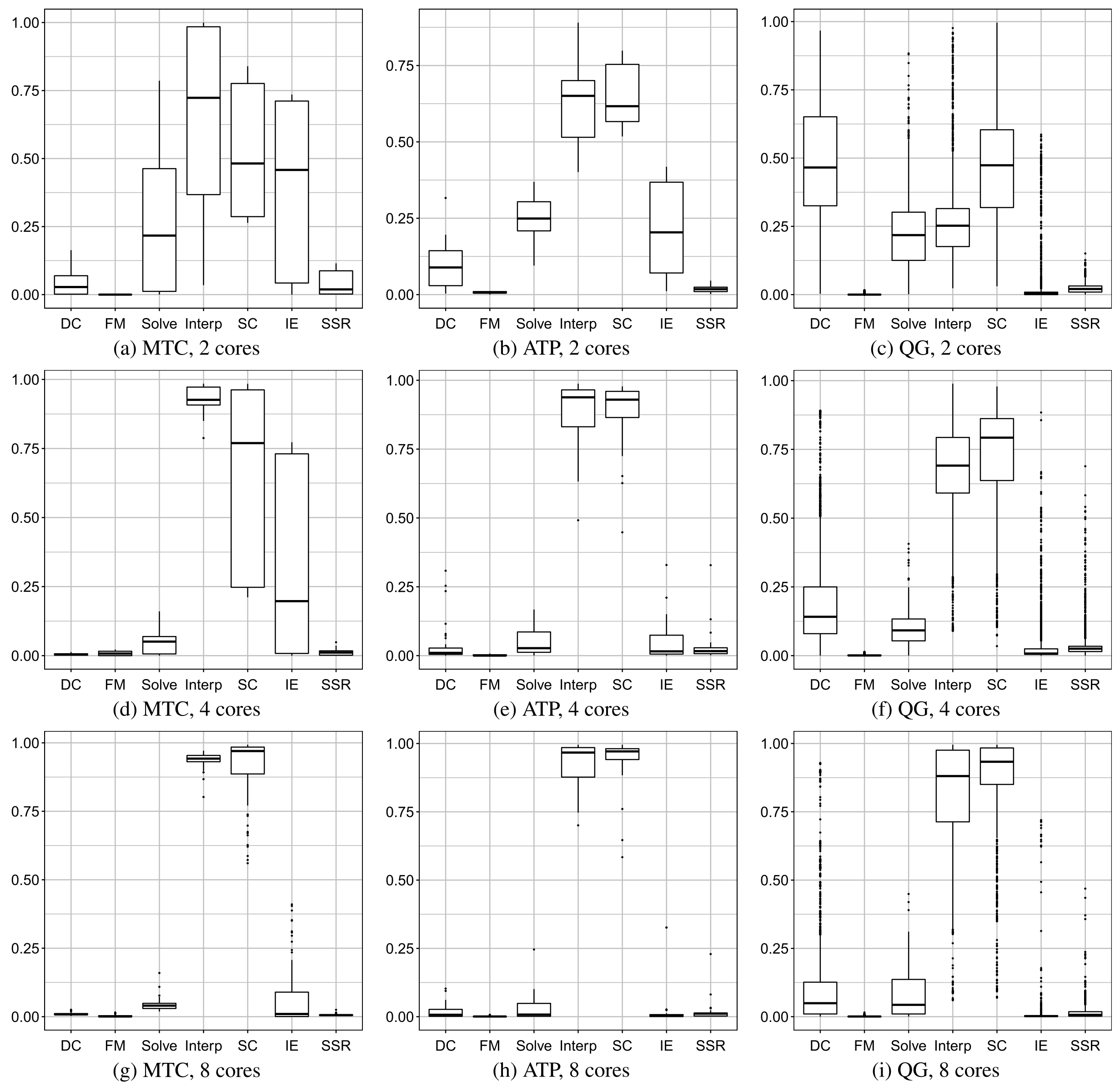
#core	MTC (%)					ATP (%)					QG (%)				
	DC	FM	SC / `Slv`	IE / `Itp`	SSR	DC	FM	SC / `Slv`	IE / `Itp`	SSR	DC	FM	SC / `Slv`	IE / `Itp`	SSR
2	4.35	0.00	54.44 / 28.95	37.30 / 62.80	3.90	9.88	0.06	88.04 / 84.46	1.86 / 5.44	0.16	47.27	0.03	50.70 / 40.74	1.07 / 11.03	0.93
	4.20	0.00	52.31 / 25.40	39.37 / 66.29	4.12	10.33	0.69	65.72 / 24.54	21.41 / 62.59	1.85	48.01	0.06	47.02 / 22.63	2.63 / 27.01	2.29
4	0.50	0.85	65.30 / 5.65	32.22 / 91.87	1.13	13.40	0.06	83.90 / 54.35	1.62 / 31.17	1.03	34.89	0.06	62.18 / 35.77	1.48 / 27.88	1.40
	0.45	0.86	64.86 / 4.38	32.68 / 93.16	1.15	3.63	0.17	88.70 / 4.86	4.59 / 88.42	2.92	19.76	0.15	73.13 / 9.17	3.57 / 67.53	3.39
8	1.03	0.26	91.68 / 5.14	6.42 / 92.96	0.61	21.27	0.03	77.51 / 54.65	0.69 / 23.55	0.50	21.50	0.04	77.48 / 30.13	0.34 / 47.69	0.64
	0.95	0.27	91.68 / 4.26	6.48 / 93.90	0.61	1.93	0.12	93.27 / 3.33	2.70 / 92.64	1.98	9.03	0.07	89.18 / 6.59	0.60 / 83.19	1.12

rows in Fig. 9 provide two profiling bases in terms of actual CPU time which excludes the CPU time wasted by thread idleness. They are based on LDC interfaces and basic operations, respectively. The actual CPU time for the conciliation step is roughly the summation of that of `Interpolate`, `Formulate`, `ComputeSharedInterp`, `CombineInterp` and `RefineSharedInterp`. The LDC procedure roughly consists of 5 kinds of basic operations: decomposition, satisfiability checking, interpolant extraction, formulation and shared symbol reasoning. It is noteworthy that the procedure of `Interpolate` consists of satisfiability checking and optional interpolant extraction depending on whether the prior checking result is unsatisfiable.

Experimental design. We prepare two special versions of PZ3 for profiling analysis: one outputs wall time for each working phase while another outputs actual CPU time for each LDC interface and basic operation. The experiments are conducted on the `QF_UF` benchmarks using 2, 4 and 8 cores.

Results and discussion. Table 8 and Table 9 report the profiling results in terms of wall time and actual CPU time, respectively. Only the problems upon which PZ3 terminates within the default timeout (600 s) are included in the profiling analysis. The percentages of wall time for each working phase and actual CPU time for each LDC interface and basic operation are computed for each problem, and the average percentages over each problem category are included in the tables. The reason is that the problems with large time costs should be prevented from substantially influencing the profiling results. Fig. 10 illustrates the distribution of actual CPU time for each LDC interface and basic operation over the problems with conciliations. Note that the problems without conciliation (i.e. NC problems) are excluded because they could bring biases to the distribution results.

In the most cases (except the case in the QG category when 2 cores are used), `Interpolate` makes a dominant contribution to the total actual CPU time, which is consistent with the observation from Table 8 that subproblem solving is the most time-expensive working phase and also implies that the conciliation step is the main bottleneck of PZ3's performance. As the number of cores grows, `Interpolate` generally takes an increasing percentage of actual CPU time because more conciliation iterations are generally required as the number of shared symbols increases. It is noteworthy that satisfiability checking plays a crucial role in `Interpolate` based on the comparison results between the time percentages of



(a) MTC, 2 cores (b) ATP, 2 cores (c) QG, 2 cores
(d) MTC, 4 cores (e) ATP, 4 cores (f) QG, 4 cores
(g) MTC, 8 cores (h) ATP, 8 cores (i) QG, 8 cores

Fig. 10. The distribution of the average percentages of actual CPU time for LDC interfaces and basic operations. DC, FM, SC, IE, SSR refer to decomposition, formulation, satisfiability checking, interpolant extraction and shared symbol reasoning, respectively. `Solve` and `Interpolate` are denoted by Solve and Interp, respectively. The caption of each sub-plot reports the problem category and the number of cores in use.

`Interpolate` and interpolant extraction in Table 9. This is because an interpolant extraction is not performed when the prior satisfiability check returns a satisfiable result.

From the perspective of basic operations, satisfiability checking is generally the most time-consuming. Fig. 10 shows that when the number of cores grows, the average time percentage of `Solve` decreases while the average time percentage of satisfiability checking increases. This is because a more fine-grained decomposition (1) reduces the difficulties of subproblems; (2) leads to more conciliation iterations and thus a larger number of satisfiability checking operations, as the number of shared symbols possibly increases. In general, decomposition has low runtime overhead. The time cost of decomposition has connections with the number of clauses N in the input problem and the size of decomposition k, as (1) clause graph construction has the worst-case time complexity $\mathcal{O}(N^2)$, and (2) extracting k subproblems has the worst-case time complexity $\mathcal{O}(kN^2)$, based on the discussion in Section 5.1. In the QG category, decomposition is substantially more time-consuming because a large portion of QG problems contain an extensive number of clauses. The profiling results also show that shared symbol reasoning and formulation have limited contributions to the total actual CPU time, and their time percentages have no strong associations with the number of cores in use.

Table 10
The design of PCVC4's portfolio. The last four options have no values, thus we use ✓ or ✗ to denote whether a solver instance (core) has the certain flag set or not.

Options	Details	core 0	core 1	core 2	core 3
`--random-freq`	frequency of random decisions	0.5	0.0	0.0	0.0
`--random-seed`	random seed	20	30	40	50
`--restart-int-base`	base restart interval	25	25	50	25
`--restart-int-inc`	restart interval increase factor	3.0	3.0	3.0	5.0
`--condense-function-values`	condense models for functions rather than explicitly representing them	✓	✗	✗	✗
`--symmetry-breaker`	use UF symmetry breaker [37]	✗	✓	✗	✗
`--uf-ss-eager-split`	add splits eagerly for UF strong solver	✗	✗	✓	✗
`--uf-ss-simple-cliques`	always use simple clique lemmas for UF strong solver	✗	✗	✗	✓

By comparing the profiling results on wall time (Table 8) and actual CPU time (Table 9), it can be observed that for the decomposition operation, its wall time ratio is generally lower than its ratio of actual CPU time. This is because the workloads of decomposition are well balanced over each working thread. Nevertheless, since the workloads of shared symbol reasoning are always assigned to the coordinator thread only while other threads are blocked, the wall time ratio of shared symbol reasoning is substantially higher than the corresponding ratio of actual CPU time.

Summary. PZ3's wall time is mostly contributed to by the subproblem solving phase. In terms of actual CPU time, `Interpolate` is the main contributor in the LDC interfaces while satisfiability checking is the main contributor in 5 kinds of basic operations.

7.7. *Comparison with the portfolio approach*

Basic concepts. The state-of-the-art technique for parallel SMT solving is the portfolio approach [33]. A portfolio is a set of solvers with different heuristics, and their combination represents a set of orthogonal yet complementary strategies. Each solver instance handles the input problem separately and its derived lemmas are shared with other solver instances. Lemma sharing could drastically improve the performance of the solver portfolio over its individual components. In what follows, we compare our approach with the portfolio approach.

Experimental design. Since CVC4 [15] has an implementation of portfolio-based parallel SMT solver named PCVC4 as the competition contribution for SMT-COMP, we compare PZ3's parallel efficiency with PCVC4's using 4 cores on the `QF_UF` benchmarks. Table 10 summarizes the design of the PCVC4's portfolio. Solver instances (cores) have orthogonal configurations on random seed, random decision, restart policy and options related to uninterpreted functions (rows 6–9). The optimal portfolio is tuned by some pilot experiments. The maximum size of shared lemma is set to 8, which is consistent with the clause sharing policy employed by ManySAT [38].

Results and discussion. Fig. 11b shows the overall comparison between PCVC4 and CVC4. On the one hand, PCVC4 fails to outperform CVC4 on a large portion of problems because (1) they have negligible benefits from the portfolio and lemma sharing, and (2) lemma sharing introduces overhead by inter-thread communications. On the other hand, PCVC4 has good performance stability, as it would not be much slower than CVC4 in the worst case. This is because the workload of each solver instance consists of solving the input problem and lemma sharing, while the latter has much less time cost than the former in the most cases.

The comparison results between PZ3 and PCVC4 are illustrated in Fig. 11. Overall, PZ3's speed-up ratio and PCVC4's are 1.593 and 0.978, respectively. To guarantee the fairness of comparison, we (1) exclude the problems in the MTC category because CVC4 can efficiently solve them by exploiting property of contradictory cycle [34]; (2) omit the problems solved within 300 ms by both sequential and parallel solvers. Then the results show that (1) PZ3 succeeds in accelerating the solutions of 1726 out of 3265 (52.9%) problems while PCVC4 accelerates the solutions of 450 out of 2267 (19.9%) problems; (2) PZ3's speed-up ratio and PCVC4's are 0.9822 and 0.9820, respectively. Therefore, PZ3 has slightly better speed-up ratio and is capable to accelerate the solutions of a substantially larger portion of problems than PCVC4.

Summary. Overall, PZ3 achieves better speed-up on the `QF_UF` benchmarks. After excluding some problems that may hinder the fairness of comparison, two parallel solvers have almost the same speed-up ratios while PZ3 can accelerate the solutions of a larger portion of problems.

8. Related work

SMT typically benefits from advances in SAT. There are generally four kinds of approaches for parallelizing SAT solving. The first is the guiding-path approach which parallelizes search in problem space based on dynamic space division.

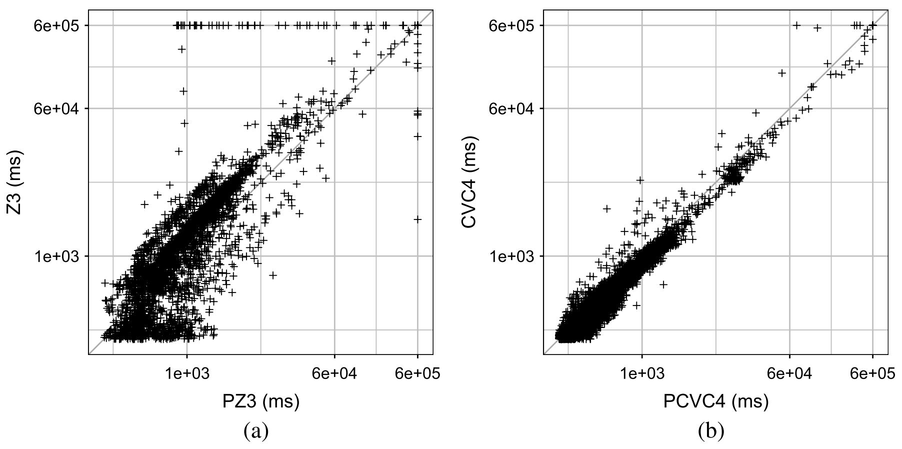


Fig. 11. Comparison of different parallelization approaches. The left scatter plot illustrates the speed-up of PZ3 over Z3, while the right one illustrates the speed-up of PCVC4 over CVC4.

Related solvers include PSATO [39], TREENGELING [40], etc. The main drawbacks of this approach are twofold. First, for a given SAT problem with hundreds of thousands of variables, it is very difficult to find the most relevant variable set for space partitioning. Second, it is challenging to partition problem space on the theory level, which is desired for lazy SMT solvers. The second is the portfolio approach, which parallelizes a solver by running multiple solver instances with different heuristics. Each solver instance handles the whole input problem separately while duplicated search is prevented by lemma sharing [41]. Solvers such as MANYSAT [38], PLINGELING [40] are based on this approach. Generally speaking, the portfolio approach is robust to various kinds of problems which are sensitive to heuristics. However, it cannot handle the problems surpassing the capability of a single solver, and a good portfolio requires elaborated parameter tuning. The third approach parallelizes unit propagation which is reported to typically take up over 80% of solving time. RISS [42] is a typical solver of this genre. This approach makes multiple decisions concurrently but handles conflicts sequentially. Empirical results show that achieved performance gains are limited because of insufficient implications and performance bottleneck on the shared clause database. The last one is the decomposition-based approach which is quite similar with LDC. This approach firstly divides the input formula into subformulae, and then conciliates interpretations of shared variables through Craig interpolation [43,44]. To the best of our knowledge, the existing decomposition-based approaches are all based on propositional logic and miss necessary mechanisms for reasoning in the theory level, such as interpretation combinability check for uninterpreted functions and refinement of interpretations for shared uninterpreted function symbols.

There are few known work on parallelizing SMT solving directly on the level of theory reasoning. Wintersteiger et al. integrate portfolio approach with lemma sharing to Z3 and the parallel solver achieves speed-up on the `QF_IDL` (quantifier-free fragment of integer differential logic) benchmarks from SMT-LIB [33]. Portfolio-based SMT solvers also suffer from the limitations of portfolio-based SAT solvers.

LDC is essentially inspired by counterexample guided abstraction refinement (CEGAR) [45–48] in program verification. Abstraction is a key to verify industrial programs as it substantially reduces the state space by removing or simplifying details irrelevant to properties to be checked. However, the information loss incurred in abstraction potentially leads to wrong results such as spurious counterexamples (an execution path that leads to the state that violates the property to be checked, but such path does not correspond to a concrete execution path). Therefore, designing abstraction is crucial and requires considerable creativity and insights. CEGAR generates an initial abstraction and refines it iteratively by analyzing the spurious counterexamples until the program is verified to be safe or a concrete counterexample is found. In [48], McMillan introduces Craig interpolation to refine the predicate abstraction. In the context of LDC, the abstraction is the global invariant, a spurious counterexample is a candidate shared interpretation inconsistent with subproblems and interpolants are used to refine the abstraction. The conciliation step is a CEGAR loop which does not terminate until a concrete counterexample (i.e. a witness of global satisfiability) is found, or no counterexamples exist (thus the input problem is unsatisfiable).

9. Conclusion and future work

In this paper, we propose a high-level and flexible framework namely lazy decomposition and conciliation (LDC) for parallelizing SMT solving. An LDC based solver could be built upon an existing SMT solver without hacking its low-level implementation. LDC is flexible for various underlying theories supporting quantifier-free interpolation. We have designed a sound and complete instantiation of LDC in $\mathcal{T}_{\mathrm{E}}$ and implemented PZ3, a parallelized version of Z3. Evaluation results substantiate the potential of LDC as PZ3 outperforms the sequential solver on a variety of random and benchmark problems. In particular, PZ3 generally takes an advantage on problems with sparse structures and it could achieve orders of magnitude speed-up over Z3 on these problems.

Our proposed framework opens some interesting directions for future research. It is possible to develop a new decomposition heuristic that takes subproblem complexity into consideration for better load balancing in parallel processing. Furthermore, it is worth instantiating LDC in other theories such as arithmetic, bit-vector and even theory combinations since they are relevant to many practical problems in verification and AI planning. Last but not the least, the implementation of LDC could be further optimized especially for dense problems. The work stealing mechanism could be integrated to improve the parallel efficiency. Some refutation compression techniques [49–52] are potentially helpful for a more efficient interpolant extraction.

Acknowledgements

This research is sponsored in part by National Natural Science Foundation of China (Grant Nos. 61402248, 61527812), National Science and Technology Major Project of China (Grant No. 2016ZX01038101), and the National Key Research and Development Program of China (Grant Nos. 2015BAG14B01-02, 2016QY07X1402).

Appendix A. Proofs for model isomorphism

Lemma 18. *$\cong$ is an equivalence relation over $\mathcal{M}$, where $\mathcal{M}$ is the set of all $\mathcal{T}_{\mathsf{E}}$-interpretations.*

Proof. We prove the reflexivity, symmetry and transitivity of $\cong$ on $\mathcal{M}$.

Reflexivity: Let $M = (\mathcal{D}, (\_)^M) \in \mathcal{M}$, and h be the identity function on $\mathcal{D}$. Then, for each constant symbol $c \in \Sigma$ we have $h(c^M) = c^M$; for each n-ary $(n > 0)$ function symbol $f \in \Sigma$ and $d_1, \dots, d_n \in \mathcal{D}$ we have $h(f^M(d_1, \dots, d_n)) = f^M(d_1, \dots, d_n)$. Hence $M \cong M$ holds.

Symmetry: Let $M_1 = (\mathcal{D}_1, (\_)^{M_1})$ and $M_2 = (\mathcal{D}_2, (\_)^{M_2})$. If $M_1 \cong M_2$ holds, there exists an isomorphism $h : \mathcal{D}_1 \to \mathcal{D}_2$ of M_1 into M_2. Since h is bijective, $h^{-1} : \mathcal{D}_2 \to \mathcal{D}_1$ is also bijective. For each constant symbol $c \in \Sigma$, $h^{-1}(c^{M_2}) = c^{M_1}$ holds. Since $h(f^{M_1}(d_1, \dots, d_n)) = f^{M_2}(h(d_1), \dots, h(d_n))$ holds for each n-ary function symbol $f \in \Sigma$ and $d_1, \dots, d_n \in \mathcal{D}_1$, we have $h^{-1}(f^{M_2}(d'_1, \dots, d'_n)) = f^{M_1}(h^{-1}(d'_1), \dots, h^{-1}(d'_n))$ where $d'_1 = h(d_1), \dots, d'_n = h(d_n)$. Hence h^{-1} is also an isomorphism of M_2 into M_1. $M_2 \cong M_1$ holds.

Transitivity: Let $M_1 = (\mathcal{D}_1, (\_)^{M_1})$, $M_2 = (\mathcal{D}_2, (\_)^{M_2})$ and $M_3 = (\mathcal{D}_3, (\_)^{M_3})$. Since $M_1 \cong M_2$ and $M_2 \cong M_3$ hold, there exists $h_a : \mathcal{D}_1 \to \mathcal{D}_2$ and $h_b : \mathcal{D}_2 \to \mathcal{D}_3$ be their isomorphisms respectively. Let h be $h_b \circ h_a$ and check if h is the isomorphism of M_1 into M_3. First, h is bijective from $\mathcal{D}_1$ to $\mathcal{D}_3$. For each constant $c \in \Sigma$, we have $h(c^{M_1}) = h_b(c^{M_2}) = c^{M_3}$. For each n-ary function $f \in \Sigma$ and $d_1, \dots, d_n \in \mathcal{D}_1$, we have $h(f^{M_1}(d_1, \dots, d_n)) = h_b(f^{M_2}(h_a(d_1), \dots, h_a(d_n)) = f^{M_3}(h(d_1), \dots, h(d_n))$. Therefore we have $M_1 \cong M_3$. □

Appendix B. Proofs for the soundness of LDC interfaces

B.1. Proof of Lemma 6

Proof. Let M be a $\mathcal{T}_{\mathsf{E}}$-interpretation, the congruence relation R of which is presented with the partition below.

$$S_\phi / R = \left\{ \{t_1^1, \dots, t_1^{k_1}\}, \dots, \{t_n^1, \dots, t_n^{k_n}\} \right\}$$

Let F be the result of `Formulate(`M`)`. $F = C_1 \wedge \cdots \wedge C_n \wedge C_{n+1}$, where

$$\begin{aligned} C_1 :&\quad t_1^1 = \cdots = t_1^{k_1} \\ &\quad\vdots \\ C_n :&\quad t_n^1 = \cdots = t_n^{k_n} \\ C_{n+1} :&\quad \bigwedge_{1 \leqslant i < j \leqslant n} t_i^1 \neq t_j^1 \end{aligned}$$

First we prove the statement: F is satisfiable and there exists only one congruence relation R over S_F that satisfies F. Clearly, we have $S_F = S_\phi$. Also, it is trivial to verify that R satisfies F. Assume that there is another congruence relation R' $(R' \neq R)$ over S_F such that R satisfies F. Without loss of generality, there exists two terms $t_a, t_b \in S_F$ such that $t_a =_R t_b$ and $t_a \neq_{R'} t_b$. Thus, by $F \to (t_a = t_b)$ and $(t_a = t_b)^{R'} = \mathsf{false}$, F is falsified by R', which contradicts to the assumption.

For an arbitrary $\mathcal{T}_{\mathsf{E}}$-formula $\varphi \in \mathcal{L}_M$, if $\varphi^M = \mathsf{true}$, we have $(\varphi \wedge F)^M = \mathsf{true}$, thus $\varphi \wedge F$ is satisfiable. Suppose $\varphi^M = \mathsf{false}$ and assume $\varphi \wedge F$ is satisfiable, then there exists an interpretation $M' \models (\varphi \wedge F)$. Since $F^{M'} = \mathsf{true}$, thus M and M' associate with the same congruence relation R over S_F. However, $\varphi^{M'} = \mathsf{true} \neq \varphi^M$, then there exists two terms $t_x, t_y \in S_\varphi$ such that $(t_x = t_y)^{M'} \neq (t_x = t_y)^M$. Thus (1) either t_x or t_y cannot be interpreted by M, implying $\varphi \notin \mathcal{L}_M$; (2) otherwise $(t_x = t_y)^{M'} = (t_x = t_y)^M$, which contradicts to our assumption. □

B.2. Proof of Lemma 7

Proof. For the first statement, we define a mapping $h : \mathcal{D} \to \mathcal{D}'$ where $\mathcal{D}$ and $\mathcal{D}'$ are domains of M and M', respectively, such that

$$h(\nu) = \begin{cases} \mu & \nu \text{ is substituted by } \mu \text{ by unification rules} \\ \nu & \text{Otherwise} \end{cases}$$

We wish to prove that h is bijective.

(1) Assume that there exists $\nu \in \mathcal{D}$ such that $h(\nu) = \mu_1$ and $h(\nu) = \mu_2$ and $\mu_1 \neq \mu_2$. Then there exists two terms t_1, t_2 such that $t_1^M = t_2^M = \nu$, $t_1^{M_S} = \mu_1$ and $t_2^{M_S} = \mu_2$, where $\mu_1 \neq \mu_2$. According to Algorithm 1, $M \models (\psi_i \wedge F_S)$ where $M_S \rhd F_S$, then we have $F_S^M = \text{true}$. By Lemma 3, $F_S \to (t_1 \neq t_2)$ holds, then $(t_1 \neq t_2)^M = \text{true}$, which contradicts to $t_1^M = t_2^M$. Hence h is a function.

(2) Analogously we can prove that there does not exist $\nu_1, \nu_2 \in \mathcal{D}$ such that $\nu_1 \neq \nu_2$ and $h(\nu_1) = h(\nu_2)$, thus h is injective. Also it is trivial that h is surjective.

By the Const-Sub and Func-Sub rules, h meets the conditions 2 and 3 in Definition 2, thus h is an isomorphism from M into M'.

For the second statement, sufficiency and necessity are proved respectively.

Sufficiency:

(1) $\nu = \nu_S$, the conclusion is trivial.

(2) ν is substituted by applying either the Const-Sub rule or the Func-Sub rule. For the former case, there exists $c \in (\Sigma_M \cap \Sigma_{M_S})$ such that $c^M = \nu$ and $c^{M_S} = \nu_S$, thus $\nu \sim \nu_S$. For the latter case, let M'' be the intermediate interpretation derived from M where the certain Func-Sub rule for the n-ary $(n > 0)$ function symbol $f \in (\Sigma_M \cap \Sigma_{M_S})$ is applicable, such that $f^{M''}(\nu_1, \ldots, \nu_n) = \nu$, $f^{M_S}(\nu_1, \ldots, \nu_n) = \nu_S$. Since M'' is derived from M with some symbols substituted, we have $f^M(\nu_1', \ldots, \nu_n') = \nu$ where $h(\nu_i') = \nu_i$ for each $1 \leq i \leq n$. By the induction hypothesis, we have $\nu_i' \sim \nu_i$, thus $\nu \sim \nu_S$ holds.

Necessity:

(1) $\nu = \nu_S$, the conclusion is trivial.

(2) There exists $c \in (\Sigma_M \cap \Sigma_{M_S})$ such that $c^M = \nu$ and $c^{M_S} = \nu_S$, then ν is substituted by ν_S by the Const-Sub rule.

(3) There exists an n-ary $(n > 0)$ function symbol $f \in (\Sigma_M \cap \Sigma_{M_S})$ such that $f^M(\nu_1, \ldots, \nu_n) = \nu$ and $f^{M_S}(\nu_{1S}, \ldots, \nu_{nS}) = \nu_S$ where $\nu_i \sim \nu_{iS}$ for each $1 \leq i \leq n$. By the induction hypothesis, we have $h(\nu_i) = \nu_{iS}$. Let M'' be the intermediate interpretation derived from M with ν_i substituted by ν_{iS} $(1 \leq i \leq n$, and it is possible that $\nu_l = \nu_{lS}$ for some $1 \leq l \leq n)$. Then, we can apply the Func-Sub rule on M'' to yield a new interpretation $M''' = M''[\nu_S/\nu]$. Hence, ν is substituted by ν_S in M', and thus $h(\nu) = \nu_S$. □

B.3. Proof of Lemma 8

Proof. It is trivial that all the function applications in F meet the conditions. Consider an arbitrary function application $(f, \mathbf{d})$ that meets the conditions. Let M_i' and M_j' be the result of `Unify(`M_i,M_S`)` and `Unify(`M_j,M_S`)`, respectively. By Lemma 7, we have $h_i(\nu_{li}) = \nu_{lS}$ and $h_j(\nu_{lj}) = \nu_{lS}$, where h_i (resp. h_j) is the isomorphism of M_i (resp. M_j) into M_i' (resp. M_j'), $\mathbf{d} = (\nu_{1S}, \ldots, \nu_{nS})$, $\mathbf{d}_i = (\nu_{1i}, \ldots, \nu_{ni})$, $\mathbf{d}_j = (\nu_{1j}, \ldots, \nu_{nj})$ and $1 \leq l \leq n$. Since both $f^{M_i}(\mathbf{d}_i)$ and $f^{M_j}(\mathbf{d}_j)$ are defined, then both $f^{M_i'}(\mathbf{d})$ and $f^{M_j'}(\mathbf{d})$ are also defined. Since $f^{M_S}(\mathbf{d})$ is not defined, by Algorithm 4 we have $(f, \mathbf{d}) \in F$. □

B.4. Proof of Lemma 9

Proof. Let M_i' be the result of `Unify(`M_i,M_S`)` for each $1 \leq i \leq k$.

Suppose `CombineInterp` returns **true**. Consider two arbitrary interpretations M_i', M_j' $(1 \leqslant i < j \leqslant k)$, an n-ary $(n > 0)$ function $f \in \Sigma_{M_i} \cap \Sigma_{M_j}$ and $\mathbf{d} \in (\mathcal{D}_i' \cap \mathcal{D}_j')^n$. If $f^{M_i'}(\mathbf{d})$ and $f^{M_j'}(\mathbf{d})$ are defined, $f^{M_S}(\mathbf{d})$ is also defined. By the Func-Sub rule, we have $f^{M_i'}(\mathbf{d}) = f^{M_j'}(\mathbf{d}) = f^{M_S}(\mathbf{d})$. Also, for each constant symbol $c \in \Sigma_{M_i} \cap \Sigma_{M_j}$, $c^{M_i'} = c^{M_j'} = c^{M_S}$ holds by the Const-Sub rule. Thus, k interpretations $M_1, \ldots, M_k$ are combinable under M_S.

Suppose `CombineInterp` returns **false**, then there exists a function application $(f, \mathbf{d})$ such that $f^{M_i'}(\mathbf{d})$ and $f^{M_j'}(\mathbf{d})$ are defined but $f^{M_S}(\mathbf{d})$ is not defined. $f^{M_i'}(\mathbf{d})$ corresponds to a term $f(\mathbf{t}_i)$ which is a subterm of $(\psi_i \wedge F_S)$ in the congruence closure, and we have $\mathbf{t}_i^{M_i'} = \mathbf{d}$. Analogously there exists $f(\mathbf{t}_j)$ which is a subterm of $(\psi_j \wedge F_S)$ such that $\mathbf{t}_j^{M_j'} = \mathbf{d}$. Furthermore, since elements in $\mathbf{d}$ are shared by M_i' and M_j', the elements in the original domains $\mathcal{D}_i$ and $\mathcal{D}_j$ are substituted by applying the Const-Sub or the Func-Sub rules. Thus, there exists (1) $(t_{i1}, \ldots, t_{in})^{M_i'} = \mathbf{d}$ where each t_{ix} $(1 \leqslant x \leqslant n)$ is built from symbols in $(\Sigma_{M_i} \cap \Sigma_{M_S})$; (2) $(t_{j1}, \ldots, t_{jn})^{M_j'} = \mathbf{d}$ where each t_{jx} $(1 \leqslant x \leqslant n)$ is built from symbols in $(\Sigma_{M_j} \cap \Sigma_{M_S})$.

Assume $M_1, \ldots, M_k$ are combinable under M_S, thus $M_1, \ldots, M_k$ have isomorphic interpretations $M''_1, \ldots, M''_k$ respectively that satisfy the condition (2) and (3) in Definition 6. By performing structural induction on term, we have

$$(t_{i1}, \ldots, t_{in})^{M''_i} = (t_{i1}, \ldots, t_{in})^{M_S} = \mathbf{d}$$
$$(t_{j1}, \ldots, t_{jn})^{M''_j} = (t_{j1}, \ldots, t_{jn})^{M_S} = \mathbf{d}$$

Also, the following two equations hold.

$$\mathbf{t}_i^{M'_i} = (t_{i1}, \ldots, t_{in})^{M'_i} = \mathbf{d}$$
$$\mathbf{t}_j^{M'_j} = (t_{j1}, \ldots, t_{jn})^{M'_j} = \mathbf{d}$$

Since $M_i \cong M'_i$ (by Lemma 7) and $M_i \cong M''_i$ (by the assumption) hold, we have $M'_i \cong M''_i$. Similarly $M'_j \cong M''_j$ holds. Thus we have $\mathbf{t}_i^{M''_i} = \mathbf{t}_j^{M''_j} = \mathbf{d}$, then $f^{M''_i}(\mathbf{d})$ and $f^{M''_j}(\mathbf{d})$ are both defined. However, $f^{M_S}(\mathbf{d})$ is not defined, which contradicts to the assumption. □

References

[1] S.O. Memik, F. Fallah, Accelerated SAT-based scheduling of control/data flow graphs, in: 20th International Conference on Computer Design, Proceedings, ICCD 2002, VLSI in Computers and Processors, Freiburg, Germany, 16–18 September 2002, IEEE Computer Society, 2002, p. 395.

[2] G. Nam, K.A. Sakallah, R.A. Rutenbar, Satisfiability-based layout revisited: detailed routing of complex FPGAs via search-based boolean SAT, in: S. Kaptanoglu, S. Trimberger (Eds.), Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays, FPGA 1999, Monterey, CA, USA, February 21–23, 1999, ACM, 1999, pp. 167–175.

[3] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, Y. Zhu, Symbolic model checking using SAT procedures instead of BDDs, in: M.J. Irwin (Ed.), Proceedings of the 36th Conference on Design Automation, New Orleans, LA, USA, June 21–25, 1999, ACM Press, 1999, pp. 317–320.

[4] R.E. Bryant, S.M. German, M.N. Velev, Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic, ACM Trans. Comput. Log. 2 (2001) 93–134.

[5] S.A. Cook, The complexity of theorem-proving procedures, in: M.A. Harrison, R.B. Banerji, J.D. Ullman (Eds.), Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, Shaker Heights, Ohio, USA, May 3–5, 1971, ACM, 1971, pp. 151–158.

[6] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient SAT solver, in: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18–22, 2001, ACM, 2001, pp. 530–535.

[7] N. Eén, N. Sörensson, An extensible SAT-solver, in: E. Giunchiglia, A. Tacchella (Eds.), Theory and Applications of Satisfiability Testing, 6th International Conference, Selected Revised Papers, SAT 2003, Santa Margherita Ligure, Italy, May 5–8, 2003, in: Lect. Notes Comput. Sci., vol. 2919, Springer, 2003, pp. 502–518.

[8] C.W. Barrett, R. Sebastiani, S.A. Seshia, C. Tinelli, Satisfiability modulo theories, in: Handbook of Satisfiability, in: Front. Artif. Intell. Appl., vol. 185, 2009, pp. 825–885.

[9] S.K. Lahiri, S.A. Seshia, The UCLID decision procedure, in: Computer Aided Verification, 16th International Conference, Proceedings, CAV 2004, July 13-17, 2004, 2004, pp. 475–478.

[10] V. Ganesh, D.L. Dill, A decision procedure for bit-vectors and arrays, in: W. Damm, H. Hermanns (Eds.), Computer Aided Verification, 19th International Conference, Proceedings, CAV 2007, Berlin, Germany, July 3–7, 2007, in: Lect. Notes Comput. Sci., vol. 4590, Springer, 2007, pp. 519–531.

[11] R. Brummayer, A. Biere, Boolector: an efficient SMT solver for bit-vectors and arrays, in: S. Kowalewski, A. Philippou (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, Proceedings, TACAS 2009, York, UK, March 22–29, 2009, in: Lect. Notes Comput. Sci., vol. 5505, Springer, 2009, pp. 174–177.

[12] L.M. de Moura, H. Rueß, An experimental evaluation of ground decision procedures, in: Computer Aided Verification, 16th International Conference, Proceedings, CAV 2004, Boston, MA, USA, July 13–17, 2004, 2004, pp. 162–174.

[13] L.M. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: C.R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Proceedings, TACAS 2008, Budapest, Hungary, March 29–April 6, 2008, in: Lect. Notes Comput. Sci., vol. 4963, Springer, 2008, pp. 337–340.

[14] B. Dutertre, L. de Moura, The Yices SMT solver, http://yices.csl.sri.com/tool-paper.pdf, 2006.

[15] C. Barrett, C.L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, C. Tinelli, CVC4, in: G. Gopalakrishnan, S. Qadeer (Eds.), Computer Aided Verification – 23rd International Conference, Proceedings, CAV 2011, Snowbird, UT, USA, July 14–20, 2011, in: Lect. Notes Comput. Sci., vol. 6806, Springer, 2011, pp. 171–177.

[16] A. Cimatti, A. Griggio, B.J. Schaafsma, R. Sebastiani, The MathSAT5 SMT solver, in: N. Piterman, S.A. Smolka (Eds.), Tools and Algorithms for the Construction and Analysis of Systems – 19th International Conference, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Proceedings, TACAS 2013, Rome, Italy, March 16–24, 2013, in: Lect. Notes Comput. Sci., vol. 7795, Springer, 2013, pp. 93–107.

[17] M. Huth, M. Ryan, Logic in Computer Science: Modeling and Reasoning about Systems, Cambridge University Press, 2004.

[18] G. Nelson, D.C. Oppen, Fast decision procedures based on congruence closure, J. ACM 27 (1980) 356–364.

[19] W. Craig, Linear reasoning. A new form of the Herbrand–Gentzen theorem, J. Symb. Log. 22 (1957) 250–268.

[20] K.L. McMillan, Interpolation and SAT-based model checking, in: W.A. Hunt Jr., F. Somenzi (Eds.), Computer Aided Verification, 15th International Conference, Proceedings, CAV 2003, Boulder, CO, USA, July 8–12, 2003, in: Lect. Notes Comput. Sci., vol. 2725, Springer, 2003, pp. 1–13.

[21] L. Kovács, A. Voronkov, Interpolation and symbol elimination, in: R.A. Schmidt (Ed.), Automated Deduction – CADE-22, 22nd International Conference on Automated Deduction, Proceedings, Montreal, Canada, August 2–7, 2009, in: Lect. Notes Comput. Sci., vol. 5663, Springer, 2009, pp. 199–213.

[22] K.L. McMillan, An interpolating theorem prover, Theor. Comput. Sci. 345 (2005) 101–121.

[23] A. Fuchs, A. Goel, J. Grundy, S. Krstic, C. Tinelli, Ground interpolation for the theory of equality, Log. Methods Comput. Sci. 8 (2012).

[24] D. Kapur, R. Majumdar, C.G. Zarba, Interpolation for data structures, in: M. Young, P.T. Devanbu (Eds.), Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5–11, 2006, ACM, 2006, pp. 105–116.

[25] A. Brillout, D. Kroening, P. Rümmer, T. Wahl, An interpolating sequent calculus for quantifier-free Presburger arithmetic, J. Autom. Reason. 47 (2011) 341–367.

[26] A. Brillout, D. Kroening, P. Rümmer, T. Wahl, Program verification via Craig interpolation for Presburger arithmetic with arrays, in: M. Aderhold, S. Autexier, H. Mantel (Eds.), 6th International Verification Workshop, VERIFY-2010, Edinburgh, UK, July 20–21, 2010, in: EPiC Ser. Comput., vol. 3, EasyChair, 2010, pp. 31–46.

[27] R. Bruttomesso, S. Ghilardi, S. Ranise, Rewriting-based quantifier-free interpolation for a theory of arrays, in: M. Schmidt-Schauß (Ed.), Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, Novi Sad, Serbia, May 30–June 1, 2011, in: LIPIcs. Leibniz Int. Proc. Inform.LIPIcs, vol. 10, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2011, pp. 171–186.

[28] A. Cimatti, A. Griggio, R. Sebastiani, Efficient generation of Craig interpolants in satisfiability modulo theories, ACM Trans. Comput. Log. 12 (2010) 7:1–7:54.

[29] K.L. McMillan, Interpolants from Z3 proofs, in: P. Bjesse, A. Slobodová (Eds.), International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30–November 02, 2011, FMCAD Inc., 2011, pp. 19–27.

[30] A. Rybalchenko, V. Sofronie-Stokkermans, Constraint solving for interpolation, J. Symb. Comput. 45 (2010) 1212–1233.

[31] G. Yorsh, M. Musuvathi, A combination method for generating interpolants, in: R. Nieuwenhuis (Ed.), Automated Deduction – CADE-20, 20th International Conference on Automated Deduction, Proceedings, Tallinn, Estonia, July 22–27, 2005, in: Lect. Notes Comput. Sci., vol. 3632, Springer, 2005, pp. 353–368.

[32] R. Bruttomesso, S. Ghilardi, S. Ranise, Quantifier-free interpolation in combinations of equality interpolating theories, ACM Trans. Comput. Log. 15 (2014) 5:1–5:34.

[33] C.M. Wintersteiger, Y. Hamadi, L.M. de Moura, A concurrent portfolio approach to SMT solving, in: A. Bouajjani, O. Maler (Eds.), Computer Aided Verification, 21st International Conference, Proceedings, CAV 2009, Grenoble, France, June 26–July 2, 2009, in: Lect. Notes Comput. Sci., vol. 5643, Springer, 2009, pp. 715–720.

[34] M. Rozanov, O. Strichman, Generating minimum transitivity constraints in p-time for deciding equality logic, Electron. Notes Theor. Comput. Sci. 198 (2008) 3–17.

[35] J. Slaney, M. Fujita, M. Stickel, Automated reasoning and exhaustive search: quasigroup existence problems, Comput. Math. Appl. 29 (1995) 115–132.

[36] W. Ackermann, Solvable Cases of the Decision Problem, Stud. Logic Found. Math., North-Holland, Amsterdam, 1954.

[37] D. Déharbe, P. Fontaine, S. Merz, B.W. Paleo, Exploiting symmetry in SMT problems, in: N. Bjørner, V. Sofronie-Stokkermans (Eds.), Automated Deduction – CADE-23 – 23rd International Conference on Automated Deduction, Proceedings, Wroclaw, Poland, July 31–August 5, 2011, in: Lect. Notes Comput. Sci., vol. 6803, Springer, 2011, pp. 222–236.

[38] Y. Hamadi, S. Jabbour, L. Sais, ManySAT: a parallel SAT solver, JSAT 6 (2009) 245–262.

[39] H. Zhang, M.P. Bonacina, J. Hsiang, PSATO: a distributed propositional prover and its application to quasigroup problems, J. Symb. Comput. 21 (1996) 543–560.

[40] A. Biere, Lingeling, Plingeling and Treengeling entering the SAT competition 2013, in: Proceedings of SAT Competition: Solver and Benchmark Descriptions, in: Dep. Comput. Sci. Ser. Publ. B, vol. B-2013-1, University of Helsinki, 2013, pp. 51–52.

[41] Y. Hamadi, S. Jabbour, L. Sais, Control-based clause sharing in parallel SAT solving, in: C. Boutilier (Ed.), Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009, Pasadena, California, USA, July 11–17, 2009, 2009, pp. 499–504.

[42] N. Manthey, Parallel SAT Solving – Using More Cores, Technical Report KRR Report 11-02, Technische Universität Dresden, 2011.

[43] Y. Hamadi, J. Marques-Silva, C.M. Wintersteiger, Lazy decomposition for distributed decision procedures, in: J. Barnat, K. Heljanko (Eds.), Proceedings 10th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2011, Snowbird, Utah, USA, July 14, 2011, in: EPTCS, vol. 72, 2011, pp. 43–54.

[44] S. Bayless, C.G. Val, T. Ball, H.H. Hoos, A.J. Hu, Efficient modular SAT solving for IC3, in: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20–23, 2013, IEEE, 2013, pp. 149–156.

[45] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement for symbolic model checking, J. ACM 50 (2003) 752–794.

[46] S. Das, D.L. Dill, Successive approximation of abstract transition relations, in: 16th Annual IEEE Symposium on Logic in Computer Science, Proceedings, Boston, Massachusetts, USA, June 16–19, 2001, IEEE Computer Society, 2001, pp. 51–58.

[47] T.A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Lazy abstraction, in: J. Launchbury, J.C. Mitchell (Eds.), Conference Record of POPL 2002: The 29th SIGPLAN–SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16–18, 2002, ACM, 2002, pp. 58–70.

[48] K.L. McMillan, Lazy abstraction with interpolants, in: T. Ball, R.B. Jones (Eds.), Computer Aided Verification, 18th International Conference, Proceedings, CAV 2006, Seattle, WA, USA, August 17–20, 2006, in: Lect. Notes Comput. Sci., vol. 4144, Springer, 2006, pp. 123–136.

[49] J. Boudou, B.W. Paleo, Compression of propositional resolution proofs by lowering subproofs, in: D. Galmiche, D. Larchey-Wendling (Eds.), Automated Reasoning with Analytic Tableaux and Related Methods – 22th International Conference, Proceedings, TABLEAUX 2013, Nancy, France, September 16–19, 2013, in: Lect. Notes Comput. Sci., vol. 8123, Springer, 2013, pp. 59–73.

[50] S. Cotton, Two techniques for minimizing resolution proofs, in: O. Strichman, S. Szeider (Eds.), Theory and Applications of Satisfiability Testing – SAT 2010, 13th International Conference, Proceedings, SAT 2010, Edinburgh, UK, July 11–14, 2010, in: Lect. Notes Comput. Sci., vol. 6175, Springer, 2010, pp. 306–312.

[51] A. Fellner, B.W. Paleo, Greedy pebbling for proof space compression, Int. J. Softw. Tools Technol. Transf. (2017), https://doi.org/10.1007/s10009-017-0459-0.

[52] J. Boudou, A. Fellner, B.W. Paleo, Skeptik: a proof compression system, in: S. Demri, D. Kapur, C. Weidenbach (Eds.), Automated Reasoning – 7th International Joint Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Proceedings, IJCAR 2014, Vienna, Austria, July 19–22, 2014, in: Lect. Notes Comput. Sci., vol. 8562, Springer, 2014, pp. 374–380.