An Algorithm to Compute Minimal Unsatisfiable Subsets for a Decidable Fragment of First-Order Formulas

Huiyuan Xie, Jie Luo
State Key Laboratory of Software Development Environment
School of Computer Science and Engineering
Beihang University, Beijing 100191, China
{xiehuiyuan,luojie}@nlsde.buaa.edu.cn

***Abstract*—In the past few years, SAT-based methods in propositional logic have been widely used to tackle practical problems in electronic design automation, software testing and hardware verification. However, lots of industrial problems can naturally be transformed to certain decidable fragments of first-order logic (FOL), which are more expressive than propositional logic. In this paper, we propose a novel algorithm to compute all minimal unsatisfiable subsets for a constrained variant of first-order formulas. By this means, we not only evaluate the satisfiability of specified formulas, but also extract minimal unsatisfiable cores. A heuristic strategy is proposed to improve the performance. Experimental results demonstrate that our algorithm performs well on many industrial instances, and the heuristic strategy is more robust than other strategies in time and space efficiency.**

I. Introduction

Lots of practical problems can be reduced to the question of determining the satisfiability for a set of first-order formulas, especially in application domains such as electronic design automation, software verification and hardware verification. A satisfiable set of formulas indicates a valid design solution. On the other hand, when the set is unsatisfiable, a need arises to locate the causes of its unsatisfiability to revise the original design accordingly. This process can be converted to the problem of extracting minimal unsatisfiable subsets (MUSes).

Given an unsatisfiable set of first-order formulas, a *minimal unsatisfiable subset* (MUS) is a subset of formulas which is unsatisfiable, but removing any one of its elements will make the remaining set satisfiable. The enumeration of MUSes is a significant and technically challenging issue in application domains ranging from model checking of software and hardware [1] to type error debugging [2], as well as in many subfields of artificial intelligence (AI), such as automated reasoning [3] and belief revision [4], [5]. In recent years, the enumeration of MUSes has been well studied (see [6]–[8]). Most implements of this problem (e.g. [9], [10]) mainly focus on propositional logic, a comparatively simple logic in current use. However, little attention has been paid to extracting MUSes in the scope of first-order logic. Since a wide variety of industrial problems can naturally be formalized to certain decidable fragments of first-order logic, conducting research on such fragments is of great importance. In this paper, the enumeration of MUSes for a function-free and equality-free fragment of first-order logic (FEF for short) is studied. The FEF fragment not only deals with simple declarative propositions, but also allows predicates and quantifiers, which makes it more expressive than pure propositional logic.

II. Preliminaries

Every FEF formula is specified in disjunctive normal form (DNF). A *DNF formula* is a disjunction of one or more clauses, and each *clause* is a conjunction of one or more literals. A *literal* is an atomic formula or its negation (denoted by '$\neg$'). The syntax of the FEF fragment is shown in Fig. 1. Note that each variable in every formula is assumed to be universally quantified. In this paper, we specify different symbols to distinguish syntax elements. Predicates are denoted by a string that begins with an uppercase letter (e.g. "Male"), while constants are denoted by a string that begins with a lowercase letter (e.g. "cat"). Variables are similar to constants, but with a question mark in the front (e.g. "?animal").

$$\begin{gathered}
\textit{DNF } formula ::= clause \vee \cdots \vee clause \\
clause ::= literal \wedge \cdots \wedge literal \\
literal ::= atomic\ formula \mid \neg\ atomic\ formula \\
atomic\ formula ::= predicate\ (term, \ldots, term) \\
term ::= variable \mid constant
\end{gathered}$$

Fig. 1. The FEF fragment of first-order logic.

Based on its syntax, the FEF fragment is a special case of effectively propositional logic (EPR), also known as the Bernays-Schönfinkel class. According to the decision procedure given by Bernays and Schönfinkel [11], the EPR fragment is decidable. Thus, the FEF fragment of first-order logic is also decidable. So it is feasible to come up with a fully automatic algorithm to compute all minimal unsatisfiable subsets for this fragment.

III. An algorithm to compute all MUSes

In this section, we first give an overview of FMUS, an algorithm for computing all MUSes for FEF formulas, followed by implementation details, optimizations and merging strategies in our algorithm.

A. Overview

Our algorithm for computing MUSes has two main features:

1) An instantiation-based method is applied.
   The basic idea behind instantiation-based method is to instantiate formulas into instances, compute all MUSes of these instances, and then map these instances back to their corresponding FEF formulas. In FMUS, instead of generating all potential instances at the very beginning, formulas are dynamically instantiated when necessary to prune the search space.
2) A constructive "split-merge" approach is adopted.
   Different from most current SAT solvers which check satisfiability through variable assignments, our approach is based on the inconsistency of clauses from the point of logical deduction [12]. For simple and clear presentation, we use $\mathcal{M}(\Gamma)$ to denote the set of MUSes of Γ. Given a DNF formula $A_1 \vee \cdots \vee A_n$ (clause A_i is a conjunction of one or more literals, where $i \in [1, n]$) and an unsatisfiable formula set $\Gamma = \Gamma' \cup \{A_1 \vee \cdots \vee A_n\}$, the computation of $\mathcal{M}(\Gamma)$ can be divided into two parts based on whether an MUS contains $A_1 \vee \cdots \vee A_n$ or not. As stated in [13], the MUSes in $\mathcal{M}(\Gamma)$ that contain $A_1 \vee \cdots \vee A_n$ can be obtained by analyzing $\mathcal{M}(\Gamma', A_1), \ldots, \mathcal{M}(\Gamma', A_n)$ respectively. To be exact, these MUSes are included in $\mathcal{M}_{A_1 \vee \cdots \vee A_n} = \{\bigcup(\Phi_i - \{A_i\}) \cup \{A_1 \vee \cdots \vee A_n\} \mid \Phi_i \in \mathcal{M}(\Gamma', A_i)$ and $A_i \in \Phi_i, where\ 1 \leq i \leq n\}$, originated from the point of logical deduction. For MUSes that have no intersection with $A_1 \vee \cdots \vee A_n$, they are included in $\mathcal{N}_{A_1 \vee \cdots \vee A_n} = \mathcal{M}(\Gamma')$. After computing $\mathcal{M}_{A_1 \vee \cdots \vee A_n}$ and $\mathcal{N}_{A_1 \vee \cdots \vee A_n}$, $\mathcal{M}(\Gamma)$ can be obtained by extracting the minimal elements of $\mathcal{M}_{A_1 \vee \cdots \vee A_n} \cup \mathcal{N}_{A_1 \vee \cdots \vee A_n}$.
   Based on the above discussion, to compute MUSes, we first need to split formulas into clauses and find all inconsistent pairs of split clauses. These clause pairs indicate the atomic contradictory relations among formulas. While iteratively merging these clauses to their original formulas one by one, the inconsistent pairs related to them are gradually merged into larger sets. Note that these sets maintain their unsatisfiability during this procedure. After all formulas are merged, a set which contains unsatisfiable subsets of the original formula set is obtained, and all MUSes are inside this set. Then we operate on this set to extract all minimal elements (i.e. all MUSes).

B. Implementation details of FMUS

As mentioned before, the general idea of FMUS is to first split formulas into clauses, then enumerate all inconsistent pairs of these clauses, and iteratively merge these pairs to MUSes of the original formula set. The distinction between propositional logic and FEF fragment is that the latter allows predicates and universally bounded variables. So we need to recall several concepts in first-order logic such as *substitution* and *most general unifier* to fully demonstrate the procedure of FMUS.

Definition 1 (Substitution)**.** In first-order logic, a *substitution* is a total mapping $\sigma : V \rightarrow T$ from variables to terms. If $?x_1, \ldots, ?x_k$ are variables in formula C, the notation $[t_1/?x_1, \ldots, t_k/?x_k]$ refers to a substitution replacing each variable $?x_i$ with a corresponding term t_i (for $i = 1, \ldots, k$), and $?x_i$ must be pairwise distinct. The result of applying a substitution list σ to a formula C is called an *instance* of C, denoted by (C, σ).

Note that in a substitution $\sigma[t/?x]$, term t is not restricted to constants. In other words, $?x$ can be replaced by either a constant or a different variable. This is crucial to reduce the number of instances and prune the search space.

Definition 2 (Most general unifier)**.** A substitution σ is a *most general unifier* (MGU) of two atomic formulas C_1 and C_2 if σ unifies them, i.e. $(C_1, \sigma) = (C_2, \sigma)$, and for any unifier σ' of these two formulas, there exists a substitution ω such that $\sigma' = \omega \circ \sigma$.

The idea is that σ is more general than any other unifiers, that is, any one of other unifiers can be obtained by combining most general unifier and a substitution. Note that there can be more than one most general unifier, but such substitutions are the same except for variable renaming.

The procedure of finding an MGU, which is called *unification*, is a pivotal part of FMUS. Unification is a "pattern matching" procedure that takes two atomic formulas as input, and returns an MGU σ if they match, or "false" if they do not match. In this paper, unification operation is implemented on the basis of the classical algorithm originated from Martelli and Montanari [14].

Definition 3 (Common substitution)**.** Given a list of substitutions $\lambda_1, \ldots, \lambda_m$, a substitution λ is a *common substitution* of $\lambda_1, \ldots, \lambda_m$ if for all $1 \leq i \leq m$, there exists a substitution ω_i such that $\lambda = \omega_i \circ \lambda_i$.

Definition 4 (Most general common substitution)**.** A common substitution λ of $\lambda_1, \ldots, \lambda_m$ is a *most general common substitution* (MGCS) if for any common substitution λ' of $\lambda_1, \ldots, \lambda_m$, there exists a substitution ω such that $\lambda' = \omega \circ \lambda$.

In FMUS, MGUs are generated when enumerating inconsistent pairs of split clauses. For two clauses that are potentially contradictory (e.g. $A(t_1, \ldots, t_k)$ and $\neg A(t_1^{'}, \ldots, t_k^{'})$), if they have an MGU σ, we call them "unifiable" and attach σ to these two clauses. Note that for clauses that have multiple literals, the inconsistent relation of clauses is analyzed by comparing their literals. For instance, given two DNF formulas $(\neg A(a) \wedge C(a)) \vee B(b)$ and $A(a) \vee D(?x)$, clause $\neg A(a) \wedge C(a)$

and clause $A(a)$ are inconsistent because of the contradiction of literal $\neg A(a)$ and literal $A(a)$ (i.e. clause $A(a)$). Every inconsistent pair is composed of two contradictory clauses and their corresponding MGU. MGUs are stored for later use at the time of clause merging. During the merging process, we check the attached MGUs of small unsatisfiable subsets to decide whether they can be merged. If there exists an MGCS of these MGUs, we merge these subsets to a larger set and update related MGUs with their MGCS; otherwise they cannot be merged and we proceed with other merging options.

Based on the discussion above, the basic steps of FMUS are listed below.

1) Given an unsatisfiable set of FEF formulas, preprocess these formulas first. Rename overlapping bound variables to eliminate name ambiguity, and then split formulas into clauses.
2) Traverse the set of split clauses, and construct all inconsistent pairs that consist of two contradictory clauses from different formulas. While extracting contradictory clauses, perform unification operation and record the corresponding MGU.
3) Before merging clauses to original formulas, analyze and process contradictory sets to filter out formulas that cannot be properly merged. In this way, we can avoid unnecessary calculation and vastly prune the search space.
4) After all the preparations above, now we come to the crux of FMUS—the merge operation. Clauses that come from the same formula are recurrently merged to their original formula through specific order[1]. When merging clauses that are originated from the same formula, we need to check whether they are unifiable, or in other words, whether the MGUs attached to these clauses can be unified to an MGCS. If they can be unified, we combine existing unsatisfiable subsets related to these clauses into a larger set and update their MGUs with the calculated MGCS; otherwise, they cannot be merged and we move on to other options.
5) After merging, we get a set where every element is an unsatisfiable subset of the input. Then all MUSes can be obtained by extracting the minimal ones from this set.

The pseudo-code for FMUS is shown in Algorithm 1. FMUS takes as input an unsatisfiable set Γ of FEF formulas, and outputs all MUSes of Γ. Lines 2-4 demonstrate the process of splitting formulas into clauses. Every formula C^i in Γ is a DNF formula $C^i_1 \vee \cdots \vee C^i_{m_i}$ where m_i stands for the number of clauses in C^i. Lines 5-15 enumerate all inconsistent pairs among split clauses to construct M_0. Specifically, lines 5-10 extract clauses that contain inconsistent literals inside themselves[2]. And lines 11-15 find all inconsistent pairs among different clauses from different formulas. The operation of `Unifiable()` is to check whether two formulas have the same structure and can be unified by applying a certain substitution (i.e. an MGU). The operation of `Unify()` returns such an MGU which is feasible to unify two formulas. The fourth loop is the most interesting but bewildering part of FMUS. In this loop, we iteratively merge formulas that contain multiple clauses. In each iteration, clauses from a certain formula are merged to the original form and the unsatisfiable subsets that contain these clauses are merged to larger unsatisfiable sets. Each round of iteration is based on the result of the previous iteration. To give a clearer explanation, let us suppose that the *i*th formula (i.e. C^i) is going to be merged and M_{i-1} is the result of the last iteration. So formula C^1 to C^{i-1} have already been merged, and formula C^i to C^n still appear in the form of split clauses. N_i is generated by extracting elements from M_{i-1} which have no intersection with C^i (Line 19). In other words, N_i stores unsatisfiable subsets which do not need to be processed in the current iteration. Relatively, S_i is a set of m_i-tuples that present all merging options with respect to C^i (Line 20). The jth item in each tuple $(\Phi^i_1, \ldots, \Phi^i_{m_i})$ is supposed to be an element of M_{i-1} that contains clause (C^i_j, σ^i_j). Then M'_i is constructed through merging all alternative $\Phi^i_1, \ldots, \Phi^i_{m_i}$ (Line 23). As a result, N_i consists of unsatisfiable subsets where C^i is not included, while M'_i is formed from unsatisfiable subsets with C^i on the inside. The operation of `MS()` is to obtain those minimal elements under set inclusion. That is, if $\Theta = \{\Theta_1, \ldots, \Theta_n\}$, where $\Theta_1, \ldots, \Theta_n$ are different sets, then $\texttt{MS}(\Theta) = \{\Theta' \mid \Theta' \in \Theta \; and \; there \; is \; no \; \Theta'' \in \Theta \; such \; that \; \Theta'' \subset \Theta'\}$. When all iterations are completed, we get M_n, the set that contains all MUSes and their corresponding MGUs. After the last loop (Lines 28-32), all MUSes are extracted.

Since the input set is composed of finite formulas, and the number of intermediate results generated during the procedure of FMUS is also finite, so FMUS must terminate in finite steps. The output of FMUS will be the set which consists of all MUSes of the input.

A simple example is presented to demonstrate the process of FMUS.

Example 1. Let $\Gamma = \{A(a), (\neg A(?x) \wedge C(a)) \vee B(?x), \neg B(b), \neg B(a)\}$. It is easy to verify that Γ is an unsatisfiable set.

For easier reading, we give an index to every formula and split them into clauses (the first loop in Algorithm 1), so we get a set of split clauses

$$C = \{\overset{[1]}{A(a)}, \overset{[2-1]}{\neg A(?x) \wedge C(a)}, \overset{[2-2]}{B(?x)}, \overset{[3]}{\neg B(b)}, \overset{[4]}{\neg B(a)}\}.$$

Since there is no element in C that contains paradoxical literals like $A(t_1, \ldots, t_k) \wedge \neg A(t'_1, \ldots, t'_k)$, we skip to the third loop statement of FMUS.

The third loop is to enumerate all inconsistent pairs of split clauses and record their corresponding MGUs. After traversing

[1]The order of merging is discussed in Section III-D.

[2]In this case, the inconsistent "pair" contains only one element.

Algorithm 1: FMUS(Γ)

Input: Γ is an unsatisfiable set of n formulas
i.e. $\Gamma = \{C^1, \ldots, C^n\}$
Output: the set of all MUSes of Γ

1 $M_0 := \emptyset$;
2 **for** $i = 1$ **to** n **do**
3 split C^i to $\{C^i_1, \ldots, C^i_{m_i}\}$;
4 $C := \bigcup_{i=1}^{n} \{C^i_1, \ldots, C^i_{m_i}\}$;
5 **forall** $C^{i_0}_{j_0} \in C$ **do**
6 **if** $C^{i_0}_{j_0}$ *contains paradoxical literals like* $A(t_1, \ldots, t_k) \wedge \neg A(t'_1, \ldots, t'_k)$ **then**
7 **if** Unifiable$(A(t_1, \ldots, t_k), \neg A(t'_1, \ldots, t'_k))$ **then**
8 $\sigma^{i_0}_{j_0} :=$ Unify$(A(t_1, \ldots, t_k), \neg A(t'_1, \ldots, t'_k))$;
9 $M_0 := M_0 \cup \{(C^{i_0}_{j_0}, \sigma^{i_0}_{j_0})\}$;
10 $C := C - \{C^{i_0}_{j_0}\}$;
11 **forall** $C^{i_1}_{j_1}, C^{i_2}_{j_2} \in C$ *where* $i_1 \neq i_2$ **do**
12 **if** $C^{i_1}_{j_1}$ *and* $C^{i_2}_{j_2}$ *are inconsistent* **then**
13 **if** Unifiable$(C^{i_1}_{j_1}, C^{i_2}_{j_2})$ **then**
14 $\sigma^{i'}_{j'} :=$ Unify$(C^{i_1}_{j_1}, C^{i_2}_{j_2})$;
15 $M_0 := M_0 \cup \{(C^{i_1}_{j_1}, \sigma^{i'}_{j'}), (C^{i_2}_{j_2}, \sigma^{i'}_{j'})\}$;
16 **for** $i = 1$ **to** n **do**
17 **if** $m_i > 1$ **then**
18 $M'_i := \emptyset$;
19 $N_i := \{\phi \mid \phi \in M_{i-1}$ *and* $\phi \cap \{C^i_1, \ldots, C^i_{m_i}\} = \emptyset\}$;
20 $S_i := \{(\Phi^i_1, \ldots, \Phi^i_{m_i}) \mid \Phi^i_j \in M_{i-1}, (C^i_j, \sigma^i_j) \in \Phi^i_j, j \in [1, m_i]\}$;
21 **forall** $(\Phi^i_1, \ldots, \Phi^i_{m_i}) \in S_i$ **do**
22 **if** $\sigma^i_1, \ldots, \sigma^i_{m_i}$ *has an MGCS* σ' **then**
23 $M'_i := M'_i \cup \{\{(C^i, \sigma')\} \cup \bigcup_{j=1}^{m_i} (\Phi^i_j - \{(C^i_j, \sigma^i_j)\})\}$;
24
25 $M_i :=$ MS$(N_i \cup M'_i)$;
26 **else**
27 $M_i := M_{i-1}$
28 **forall** $\Theta \in M_n$ **do**
29 $\Theta' := \emptyset, M' := \emptyset$;
30 **forall** $(C, \sigma) \in \Theta$ **do**
31 $\Theta' := \Theta' \cup \{C\}$;
32 $M' := M' \cup \{\Theta'\}$;
33 **return** *M';*

set C, we get

$$M_0 = \big\{\{(A(a), \overset{[1]}{[a/?x]}), (\neg A(?x) \wedge \overset{[2-1]}{C(a)}, [a/?x])\},$$
$$\{(B(?x), \overset{[2-2]}{[b/?x]}), (\neg B(b), \overset{[3]}{[b/?x]})\},$$
$$\{(B(?x), \overset{[2-2]}{[a/?x]}), (\neg B(a), \overset{[4]}{[a/?x]})\}\big\}.$$

All inconsistent pairs of split clauses are stored in M_0.

Then we proceed with the forth loop statement. Since formula 1 is an atomic formula (i.e. $m_1 = 1$), we have

$$M_1 = M_0.$$

Then we continue to merge formula 2, where $m_2 = 2$:

It is obvious that every element in M_1 contains a clause from formula 2, so N_2 is empty. Because formula 2 has two clauses, every element in S_2 is a 2-tuple (Φ_1, Φ_2) where Φ_1 is an element of M_1 that contains clause $\neg A(?x) \wedge C(a)$ and Φ_2 is an element of M_1 that contains clause $B(?x)$. Therefore, S_2 has two elements:

$$S_2 = \big\{(\Phi_1, \Phi_2), (\Phi'_1, \Phi'_2)\big\}, \textbf{where}$$
$$\Phi_1 = \Phi'_1 = \{(A(a), \overset{[1]}{[a/?x]}), (\neg A(?x) \wedge \overset{[2-1]}{C(a)}, [a/?x])\},$$
$$\Phi_2 = \{(B(?x), \overset{[2-2]}{[b/?x]}), (\neg B(b), \overset{[3]}{[b/?x]})\},$$
$$\Phi'_2 = \{(B(?x), \overset{[2-2]}{[a/?x]}), (\neg B(a), \overset{[4]}{[a/?x]})\}.$$

M'_2 is constructed from S_2. For (Φ_1, Φ_2), the attached MGU of Φ_1 (i.e. $[a/?x]$) and the attached MGU of Φ_2 (i.e. $[b/?x]$) are not unifiable, so Φ_1 and Φ_2 cannot be merged. But for (Φ'_1, Φ'_2), since they share the same MGU, they can be merged to a larger set according to Line 23 in Algorithm 1. Thus we have

$$M'_2 = \big\{\{(A(a), \overset{[1]}{[a/?x]}), ((\neg A(?x) \wedge C(a)) \overset{[2]}{\vee} B(?x), [a/?x]),$$
$$(\neg B(a), \overset{[4]}{[a/?x]})\}\big\}.$$

Therefore, we get

$$M_2 = \texttt{MS}(M'_2 \cup N_2) = M'_2.$$

After the above steps, clauses from formula 2 are merged to the original form, and inconsistent pairs are combined to larger unsatisfiable sets (with formula 2 on the inside).

Because formula 3 and formula 4 are atomic formulas, we have

$$M_4 = M_3 = M_2.$$

After the last loop, we have

$$M' = \big\{\{\overset{[1]}{A(a)}, (\neg A(?x) \wedge \overset{[2]}{C(a)}) \vee B(?x), \overset{[4]}{\neg B(a)}\}\big\}.$$

Therefore, the set of all MUSes of Γ is M'.

C. Optimization

The above algorithm presents a basic form of the fundamental ideas to compute MUSes. In order to generate all MUSes, we need to take all possible options into account and record every potential intermediate result. So at the early stage of merging, the quantity of unsatisfiable clause sets increases rapidly, leading to immense consumption of memory and runtime. To achieve better performance, some additions and optimizations were made to rein in the explosive growth of unsatisfiable clause sets and eliminate redundant intermediate results.

1. We process the original formulas before merging to filter out clauses that can not be part of any minimal unsatisfiable set, which helps to reduce unnecessary merging operations to expedite the execution time.

2. In the process of merging, we compare the union of alternative sets that are going to be merged with MUSes that we have obtained. If there already is an MUS which is a subset of the union set, it indicates that these sets can never be merged to an MUS, so we rule out this alternative to prune the computing space. In much the same way, we sift through the intermediate results after each round of the mergence to eliminate redundant results.

3. The third major optimization is the use of bitset. This provides a compact way to store a set of items and conditions. In our algorithm, after splitting formulas and extracting contradictory clauses, we no longer concern with the specific predicates and variables, but only their inconsistent situations and relevant substitutions (MGUs). So instead of storing all clauses in basic data structures such as array and set, we use bits to represent the appearance of certain clauses in unsatisfiable sets. By using this compact representation, a significant reduction in memory consumption can be achieved. Besides, the query and comparison of set elements can be accomplished by using simple and fast bit operations, which improves time efficiency as well.

D. Merging strategies

For FMUS, the most time-consuming part is the merging operation. Although the order of merging does not affect correctness, it does affect the number of intermediate results as well as the convergence rate. A good merging strategy should decrease the peak memory consumption, and reduce runtime in the meantime. In this paper, we propose a heuristic merging strategy and compare its efficiency with sequential merging and random merging.

- *Sequential merging* depends on the original input order which is defined artificially. In this case, empirical human involvement is allowed through properly arranging the order of input formulas in advance.
- *Random merging* neglects the original input order and randomly generates merging order every time it is executed.
- *Heuristic merging* is based on the idea of restricting the increment rate of conflicting clause sets and speeding up convergence. Every clause in a formula may contradict with a plurality of clauses in other formulas. If formula C^i has m clauses, and the numbers of contradictory clauses of $C^i_1, \ldots, C^i_m$ are $n_1, \ldots, n_m$, the process of merging C^i would produce $\prod_{j=1}^{m} n_j$ intermediate unsatisfiable sets theoretically. What we want is to rein in the potentially exponential growth of intermediate results as much as possible. Thus before merging, the numbers of potential merging results for all formulas are calculated and the heuristic merging order is obtained through arranging these formulas by the calculated numbers, from least to most. To make a fair comparison, we also implement a completely opposite order as a contrast strategy.

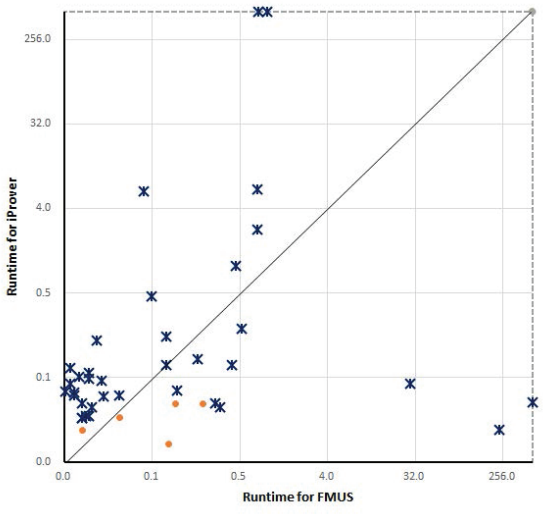

Fig. 2. Comparing FMUS against iProver on industrial benchmarks.

IV. Implementation and Experimental Results

In this section, we present empirical results demonstrating the general performance of FMUS on industrial benchmarks, as well as the effectiveness of optimizations and the heuristic strategy we adopted. All experiments were performed on a Linux server with an Intel Xeon 2.6GHz CPU and 24GB main memory. Abort timeout is 600 seconds for all test cases.

A. General performance

As mentioned earlier, the FEF fragment we focus on is a special case of EPR. Since most implements of MUS enumeration mainly deal with propositional logic, we evaluated the performance of FMUS on FEF by comparing its performance on industrial benchmarks with one of the state-of-the-art EPR solvers—iProver [15]. Evaluation was performed on the TPTP Problem Library[3] in the EPR division. The majority instances considered was originally from realistic problems, including geometry, puzzles, and software verification.

Fig. 2 shows scatter plot comparing the runtime of FMUS with the runtime of iProver for each instance. The x-coordinate of each point is the time used by FMUS, and the y-coordinate is the time used by iProver. The diagonal line $y = x$ is also plotted. In the plot, unsatisfiable instances are denoted as "*". Points above the diagonal correspond to instances on which FMUS outperforms iProver, while points below it correspond to instances on which iProver performs better. According to the comparison results, FMUS is comparable to iProver in time efficiency, with most of the points located above the diagonal. Moreover, instead of only providing a simple "unsatisfiable" response like iProver, FMUS is able to find all MUSes in the meantime, demonstrating its evident competence on dealing

[3]http://www.cs.miami.edu/~tptp/

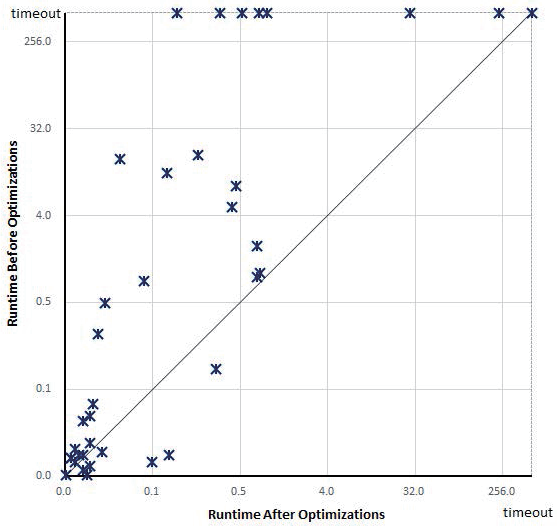

Fig. 3. Comparing runtimes before and after optimizations.

with realistic problems. We also analyzed the performance of FMUS on satisfiable instances (denoted as "·"), which are served as extreme cases since there should be no MUSes in the results. In this case, FMUS seems to slightly lose its potency when compared with iProver. A possible explanation is that FMUS aims at enumerating all MUSes, so it has to complete all iterations to give the results. Nevertheless, iProver is able to return "satisfiable" as soon as a feasible assignment is detected.

B. Effectiveness of optimizations

In order to evaluate the effectiveness of the discussed optimizations (the adoption of bitset, etc.), we selected a variety of EPR instances from the TPTP Problem Library and analyzed the runtimes before and after the optimizations. The empirical results are shown in Fig. 3. The x-coordinate presents the runtime after optimizations, while the y-coordinate presents the runtime before optimizations. The diagonal line $y = x$ is plotted as well. With few outliers, most points are above the diagonal line. We notice that the implementation with optimizations is about an order of magnitude faster than before on many instances, and it was able to solve all instances within the timeout period while the implementation without optimizations failed on several instances. Thus we can draw the conclusion that these optimizations are effective and can achieve an obvious speed-up over before.

C. Effectiveness of the heuristic strategy

We have implemented our algorithm with all merging strategies discussed in this paper. A series of experiments were performed on both industrial benchmarks as well as randomly generated benchmarks to investigate the effectiveness of the proposed heuristic strategy. Industrial instances were obtained from the PUZ subdirectory of the TPTP Problem Library. And we randomly generated benchmarks to simulate comparatively large-scale instances. Each benchmark class contains 300 instances with the same number of formulas, denoted as the form "mc$x$-y". The first number x in benchmark name stands for the number of formulas in each instance (i.e. the size of the original formula set), and the second number y stands for the average number of clauses in each instance. For example, benchmark class "mc500-1126" is composed of instances that have 500 formulas, and the average number of clauses in these instances is 1126. Although the number of formulas (i.e. x) is fixed in each benchmark class, the number of predicates and variables within formulas can vary (so y is an average number), which allows us to simulate as many cases as possible.

Table I shows evaluation results on industrial instances. The first column of Table I presents the instance name. The next two columns give the number of formulas and the number of clauses in each instance. The next group of columns list runtimes (in seconds) for different merging strategies. A notation of "-" indicates that the timeout was reached. The bold number in each row represents the shortest runtime among different strategies. As can be observed, when the number of formulas is small, the runtimes for different merging strategies are close. But when the number of formulas becomes larger and the structure of formulas becomes more complex, the heuristic strategy is clearly more efficient than the others. For example, when the number of clauses increases to 106, all merging strategies fail to give a result within the timeout except the heuristic strategy. The fundamental cause lies in the theoretical computation complexity of this problem. When the size of input increases, there can be a potentially exponential growth of intermediate results accordingly, and this is why an efficient heuristic strategy is needed. As can be concluded, employing the discussed heuristic strategy greatly improves performance on practical problems in general, which we view as a reasonable metric of its effectiveness.

Given that practical problems are relatively scattered and vary hugely in structure, randomly generated benchmarks are adopted to further evaluate the performance of the heuristic strategy on large-scale instances in a statistical way. The runtimes used by different merging strategies on randomly generated benchmarks are shown in Table II. Each benchmark class consists of 300 different instances. The total time and the average time spent on the solved instances are counted, and the number of aborted instances is shown in the parentheses after runtime. The bold numbers in each row represent the shortest total time and the shortest average time among different strategies. Note that the contrast strategy adopts an opposite strategy to the heuristic strategy.

From the experimental data we can see that different merging strategies greatly affect the performance of our algorithm. It is obvious that the heuristic strategy yields the best performance overall, especially in larger test cases. This is to be expected, because the number of MUSes can potentially be exponential in the size of input. Therefore, the prowess of the heuristic strategy is clearer when dealing with larger instances. When the number of formulas is larger than 300, the runtime of the heuristic merging is at least an order of

TABLE I
RUNTIMES FOR DIFFERENT MERGING STRATEGIES ON INDUSTRIAL BENCHMARKS

Instance	#Formulas	#Clauses	Sequential Merging	Random Merging	Heuristic Merging	Contrast Strategy
PUZ001-3	12	22	0.012	**0.011**	0.019	0.057
PUZ029-1	15	36	0.011	0.010	**0.010**	0.043
PUZ014-1	20	49	0.379	0.132	**0.017**	0.154
PUZ028-6	41	51	0.017	0.199	**0.014**	0.079
PUZ018-2	47	71	0.624	0.908	**0.411**	0.647
PUZ018-1	48	72	0.930	1.311	**0.453**	1.653
PUZ030-1	43	106	-	-	**28.466**	-
PUZ016-2.004	66	140	-	-	**0.313**	-
PUZ016-2.005	117	256	-	-	**504.209**	-

* Abort timeout is 600 seconds for every instance.

magnitude shorter than the random merging. The heuristic strategy is able to find solutions for all test cases in reasonable amount of time, while other strategies are generally aborted on most instances because of timeout when the number of formulas grows to 700. As expected, the contrast strategy has the worst performance, with the longest average runtimes and the largest number of aborted instances.

We also select a sufficient number of instances from different benchmarks to analyze the space consumption of different merging strategies. Fig. 4 demonstrates a typical comparison of how the numbers of intermediate conflicting sets for different strategies are changing while running "mc70-198-3", a test case in benchmark class "mc70-198". The x-axis represents the number of merged formulas, and the y-axis represents the number of intermediate results at each stage. More specifically, there are 43 formulas that need to be merged in this test case, thus the y-value becomes zero when the x-value increases to 43, indicating the end of the mergence; for the random merging strategy, the number of intermediate results peaks when its x-value is 19. While the contrast strategy (i.e. the opposite of the heuristic strategy) triggers a visible explosion of intermediate results at an early age, the required space for the heuristic strategy increases slowly. And the peak value of the heuristic strategy is more than an order of magnitude smaller than that of the contrast strategy, which contributes to a great reduction in memory usage. Thus as a statistical result, the heuristic order outperforms other merging strategies on most of the benchmarks, both in time efficiency and space efficiency.

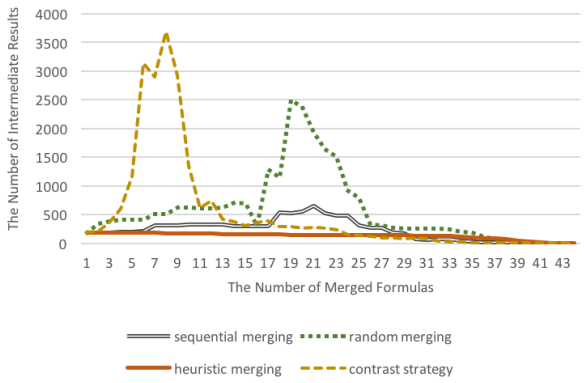

Fig. 4. Variation trends of intermediate conflicting sets for different strategies.

V. RELATED WORK

The enumeration of MUSes has been well studied and different classes of algorithms have been presented. Early algorithms are based on subset enumeration [2], [16]. In these algorithms, the power set of the input is enumerated in a tree structure and every subset is checked for satisfiability. An MUS is an unsatisfiable subset whose proper subsets are all satisfiable. Another class of algorithms [9], [17], [18] relies on the hitting set duality. In the first phase, all minimal correction subsets (MCSes) are computed; after computing minimal hitting sets of these MCSes, all MUSes are obtained. Recently, algorithms (e.g. eMUS [10] and MARCO [19]) for partial MUS enumeration were proposed. These algorithms are able to produce the first MUS quickly and early, and the following MUSes are generally produced at a steady rate.

However, there are several major differences between our approach and the algorithms mentioned above. First, the above algorithms mainly deal with Boolean satisfiability problems, while our algorithm focuses on enumerating all MUSes for the FEF fragment of first-order logic, which is more expressive than pure propositional logic.

Different from most approaches which make use of variable assignments to check satisfiability, our "split-merge" approach is driven by the atomic inconsistent relations among formulas, which is inspired by the process of logical deduction in belief revision [20], [21].

Moreover, to solve the satisfiability problem in first-order logic, many implements (e.g. [22], [23]) first transform first-order formulas to SAT instances, and then rely on existing SAT solvers to deal with these instances. To achieve a complete search process, every variable is instantiated with all values it can take. This will generate excessive instances and lead to a vast search space. Therefore, unification operation is used in our algorithm to accomplish a "general instantiation", that is to say, instead of enumerating all feasible values of variables, we update MGUs to keep track of the substitutions of variables. By this means, the search space can be reduced to a considerable extent.

TABLE II
RUNTIMES FOR DIFFERENT MERGING STRATEGIES ON RANDOMLY GENERATED BENCHMARKS

Benchmark	Sequential Merging		Random Merging		Heuristic Merging		Contrast Strategy	
	Sum	Average	Sum	Average	Sum	Average	Sum	Average
mc100-381	132.979	0.443	132.334	0.441	**18.731**	**0.062**	1907.453	6.358
mc300-623	2547.325(5)	8.635(5)	2151.36(3)	7.244(3)	**58.731**	**0.196**	12103.76(2)	40.617(2)
mc500-1126	9590.778(9)	32.958(9)	10432.12(3)	35.125(3)	**417.409**	**1.391**	41477.7(36)	157.113(36)
mc700-1544	-	-	-	-	**4358.71**	**14.529**	-	-
mc1000-3271	-	-	-	-	**10638.3**	**35.462**	-	-

* Abort timeout is 600 seconds for every instance.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a novel algorithm to compute all minimal unsatisfiable subsets for a decidable fragment of first-order logic. The computational process is based on two aspects: First, an instantiation-based method is employed; second, all minimal unsatisfiable subsets are enumerated through a constructive "split-merge" approach. According to our experimental results, our algorithm performs well on realistic problems, and the heuristic strategy as well as the adopted optimizations has proved to be effective.

For future work, we would like to analyze the exact relationship among runtime, the size of input set and the number of predicates. Our experimental results imply that there is a positive correlation between runtime and the size of the original instance, while the number of predicates and variables are negatively associated with runtime, but their quantified relevance remains unknown. Analyzing this problem in a quantitative way might shed some light on the key factors that affect algorithm performance and provide us with new ideas for optimization.

As mentioned before, merging strategy for clause mergence can significantly affect the performance of FMUS. Thus additional improvements to FMUS are expected. For instance, it would be interesting to explore better merging strategies and techniques to intelligently select an optimal strategy according to the characteristics of the input. Besides, we would like to take equality predicate into consideration in future work and investigate whether our algorithm can be applied to larger fragments of first-order logic.

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China (Grand No. 61502022) and State Key Laboratory of Software Development Environment (Grand No. SKLSDE-2015ZX-22).

REFERENCES

[1] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah, "Reveal: A formal verification tool for verilog designs," *Lecture Notes in Computer Science*, vol. 5330, pp. 343–352, 2008.

[2] M. G. D. L. Banda, P. J. Stuckey, and J. Wazny, "Finding all minimal unsatisfiable subsets," in *International ACM Sigplan Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, 2003, pp. 32–43.

[3] M. Janota and J. Marques-Silva, "cmmus : A tool for circumscription-based mus membership testing," in *International Conference on Logic Programming and Nonmonotonic Reasoning*, 2011, pp. 266–271.

[4] J. Luo and W. Li, "An algorithm to compute maximal contractions for horn clauses," *Science China Information Sciences*, vol. 54, no. 2, pp. 244–257, 2011.

[5] D. Jiang, W. Li, J. Luo, Y. Lou, and Z. Liao, "A decomposition based algorithm for maximal contractions," *Frontiers of Computer Science*, vol. 7, no. 6, pp. 801–811, 2013.

[6] F. Bacchus and G. Katsirelos, *Finding a Collection of MUSes Incrementally*, 2016.

[7] V. Ryvchin and O. Strichman, "Faster extraction of high-level minimal unsatisfiable cores," in *Theory and Applications of Satisfiability Testing - SAT 2011 - International Conference, SAT 2011, Ann Arbor, Mi, USA, June 19-22, 2011. Proceedings*, 2011, pp. 174–187.

[8] G. Xiao and Y. Ma, "Inconsistency measurement based on variables in minimal unsatisfiable subsets," *Frontiers in Artificial Intelligence & Applications*, vol. 242, 2012.

[9] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," *Journal of Automated Reasoning*, vol. 40, no. 1, pp. 1–33, 2008.

[10] A. Previti and J. Marques-Silva, "Partial mus enumeration," in *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[11] F. P. Ramsey, "On a problem of formal logic," *Proceedings of the London Mathematical Society*, vol. 30, no. 1, pp. 1–24, 2009.

[12] W. Li, "R-calculus: An inference system for belief revision," *Computer Journal*, vol. 50, no. 4, pp. 378–390, 2007.

[13] J. Luo, "A general framework for computing maximal contractions," *Frontiers of Computer Science Selected Publications from Chinese Universities*, vol. 7, no. 1, pp. 83–94, 2013.

[14] A. Martelli and U. Montanari, "An efficient unification algorithm," *ACM Transactions on Programming Languages & Systems*, vol. 4, no. 2, pp. 258–282, 1982.

[15] K. Korovin, "iprover—an instantiation-based theorem prover for first-order logic," in *Automated Reasoning, International Joint Conference, Ijcar 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, 2008, pp. 2172–2178.

[16] A. Hou, "A theory of measurement in diagnosis from first principles," *Artificial Intelligence*, vol. 65, no. 2, pp. 281–328, 1994.

[17] J. Bailey and P. J. Stuckey, "Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization," *Lecture Notes in Computer Science*, vol. 3350, pp. 174–186, 2005.

[18] R. Stern, M. Kalech, A. Feldman, and G. Provan, "Exploring the duality in conflict-directed model-based diagnosis," in *AAAI Conference on Artificial Intelligence*, 2012.

[19] M. H. Liffiton, A. Previti, A. Malik, and J. Marques-Silva, "Fast, flexible mus enumeration," *Constraints*, pp. 1–28, 2015.

[20] J. Luo and W. Li, "R-calculus without the cut rule," *Sciece China Information Sciences*, vol. 54, no. 12, pp. 2530–2543, 2011.

[21] W. Li, N. Shen, and J. Wang, "R-calculus: A logical approach for knowledge base maintenance," in *International Conference on Tools with Artificial Intelligence, 1994. Proceedings*, 1994, pp. 375–381.

[22] W. McCune, "Mace 2.0 reference manual and guide," *Mpich Working Note*, 2001.

[23] S. Kim and H. Zhang, "Modgen: theorem proving by model generation," 1997, pp. 162–167.