Computing Minimal Unsatisfiable Core for LTL over Finite Traces*

Tong Niu[1], Shengping Xiao[1], Xiaoyu Zhang[1], Jianwen Li[1,2]✉, Yanhong Huang[1,3]✉, and Jianqi Shi[1,3]

[1] East China Normal University
[2] Shanghai Key Laboratory of Trustworthy Computing
[3] National Trusted Embedded Software Engineering Technology Research Center
{jwli,yhhuang}@sei.ecnu.edu.cn

Abstract. In this paper, we consider the Minimal Unsatisfiable Core (MUC) problem for Linear Temporal Logic over finite traces (LTL_f), which nowadays is a popular formal-specification language for AI-related systems. Efficient algorithms to compute such MUCs can help locate the inconsistency rapidly in the written LTL_f specification and are very useful for the system designers to amend the flawed requirement. As far as we know, there are no available tools off-the-shelf so far that provide MUC computation for LTL_f. We present here two generic approaches NaiveMUC and BinaryMUC to compute an MUC for LTL_f. Moreover, we introduce heuristics that are based on the Boolean Unsatisfiable Core (UC) technique to accelerate the two approaches, which are named NaiveMUC+UC and BinaryMUC+UC, respectively. In particular, for global LTL_f formulas, we show that the MUC computation can be reduced to the pure Boolean MUC computation, which therefore conducts the GlobalMUC approach. Our experiments show that GlobalMUC performs the best to compute an MUC for global formulas, and BinaryMUC+UC is the best for an arbitrary unsatisfiable formula.

Keywords: Minimal Unsatisfiable Core · Linear Temporal Logic over finite traces · Boolean Unsatisfiable Core.

1 Introduction

Linear Temporal Logic over finite traces, or LTL_f, is a formal specification language that describes system behaviors in a mathematical/logical way. Basically, LTL_f is a variant of Linear Temporal Logic (LTL), which was introduced into computer science in 1977 [29] and is interpreted over infinite traces. Compared to that, LTL_f is interpreted over finite traces, which is more suitable to capture the scenarios in AI, e.g. planning [1, 10, 5, 28, 6]. Because of the wide spectrum of applications in the AI community [8], fundamental techniques for LTL_f reasoning, e.g., satisfiability checking [24, 23], the translation to automata [40, 33, 11] and synthesis [39, 2, 34, 38, 18] have been investigated in depth.

* Yanhong Huang is the corresponding author.

In this paper, we focus on another fundamental problem of LTL_f, i.e., the computation of *minimal unsatisfiable core* (MUC) for an unsatisfiable formula. In scenarios where specifications are written in some temporal logic, the satisfiability-checking process is normally provided as soon as the specification is ready. If the checking result turns out to be unsatisfiable, the specification is meaningless and has to be amended. Such procedure is called *specification debugging* and has been widely used in relevant domains [27, 12, 13, 37]. Meanwhile, specification debugging is also useful for the AI community, considering that more and more applications are formally specified by LTL_f. An MUC of an unsatisfiable formula (specification) represents a minimal part of the formula that causes unsatisfiability. Computing the MUCs can help locate the conflict in the specification efficiently and reduce the cost of specification debugging.

Inconsistency diagnosis (or finding minimal unsatisfiable cores) has been studied in AI scenarios like declarative process [35, 14], user preference [22] and configuration [16], to name a few, where the specifications (or constraints) are not described as LTL_f formulas but as regular expressions or predicates. Compared to that, this paper presents dedicated approaches to compute MUC of LTL_f formulas for further diagnosis.

There are several works on extracting the (Minimal) Unsatisfiable Cores (UCs) for LTL formulas [7, 31, 32, 20]. Hantry et al. studied the complexity of computing MUCs for LTL formulas in theory but did not give a practical implementation [20]. Cimatti et al. presented two kinds of UC-extraction methods for LTL by exploiting BDD [4] and SAT [15] techniques respectively [7], which however, both need to reduce LTL satisfiability to model checking and were shown not efficient in previous works [25, 26]. Schuppan presented a resolution-based approach to extract the UC (Unsatisfiable Core) of an unsatisfiable LTL formula [31, 32] and integrated it into the LTL satisfiability checker TRP++ [21]. The main challenge to applying such a methodology for specification debugging may be a heavy translation from the input formula to its *Separated Normal Form* (SNF) [17] has to be involved[4]. Based on the resolution method to compute UCs for LTL formulas, Goré et al. present further to compute the MUC by exploiting a single BDD computation [19].

Towards LTL_f UC (or MUC) computation, it cannot be reduced to LTL UC (or MUC) compuation directly, even though they have the same syntax. For example, $\Box \bigcirc a$ requires that at every timestamp a has to be true in the next timestamp. Such a formula is unsatisfiable in LTL_f since the last position of an arbitrary model does not have a next position, while it is satisfiable in LTL for that LTL is interpreted over infinite models. As a result, more efforts need to pay to compute LTL_f UCs or MUCs. Recently, four different approaches to compute the unsatisfiable core of LTL_f formulas are investigated in [30], three of which are extended from those for LTL UC-extraction proposed in [7, 31] and the other is based on our previous work on LTL_f satisfiability checking [23]. Yet the UCs computed by these approaches are not necessarily minimal. The main

[4] From the literature [17], the size of generated SNF can be 10X larger than the original formula.

goal of this paper is to conduct an efficient MUC solver for unsatisfiable LTL$_f$ formulas. We also show later that computing MUCs by using the approaches of this paper can be advantageous to computing UCs by the algorithms proposed in [30].

Given an unsatisfiable LTL$_f$ formula φ with the conjunctive form $\bigwedge \varphi_i$, let S_φ be the set of conjuncts of φ. We formally define one MUC *muc* of φ is a subset of S_φ such that (1) $\bigwedge \varphi_i$ is unsatisfiable for $\varphi_i \in muc$ and (2) $\bigwedge \varphi_j$ is satisfiable for each $\varphi_j \in S$ where $S \subsetneq muc$. An intuitive solution is to delete elements in S_φ one by one and check whether the deleted element has to be in the MUC. Also, to improve performance, one may adopt the dichotomy strategy to delete elements in a more aggressive way. These two trivial approaches are named *NaiveMUC* and *BinaryMUC*, respectively.

However, the above two approaches are generic and do not utilize the inherent features of LTL$_f$ for MUC computation. In this paper, we leverage two important observations from [24] and thus present the corresponding heuristics dedicated to LTL$_f$. We first utilize the concept of *obligation formula* for an LTL$_f$ formula, which essentially is a Boolean formula indicating the satisfiability of the corresponding LTL$_f$ formula. When the obligation formula is unsatisfiable, an SAT solver [15] can return a UC that relates to a subset of the original LTL$_f$ formula. If such LTL$_f$ part is still unsatisfiable, the MUC can be computed only based on this part of the formula. Therefore, the elements that need to be considered in the original formula set can be potentially reduced. We apply it to both NaiveMUC and BinaryMUC and therefore present the *NaiveMUC+UC* and *BinaryMUC+UC* approaches. Secondly, for global LTL$_f$ formulas with the form of $\Box\psi$ ($\Box$ is the global operator introduced below), we show that the MUC computation can be reduced to the Boolean MUC computation, enabling us to leverage the state-of-the-art MUC solver for Boolean formulas [3] to efficiently compute the MUC for unsatisfiable global LTL$_f$ formulas. Such an approach is named *GlobalMUC*.

We conduct an extensive experimental evaluation on the five different approaches to MUC computation for LTL$_f$. We benchmarked the tools with unsatisfiable formulas from the widely-used patterns in relevant domains [23, 27]. The results show that GlobalMUC performs best on computing MUCs for unsatisfiable global formulas. In fact, it can achieve a 300X speedup than the second-best approach BinaryMUC+UC. For general instances, BinaryMUC+UC performs the best, followed by BinaryMUC, NaiveMUC+UC, and NaiveMUC in order. In particular, BinaryMUC+UC is able to gain more than 10% performance improvement than the second-best one BinaryMUC.

In summary, the main contribution of this paper is to present different approaches for computing LTL$_f$ MUCs and investigate the best solutions through a comprehensive evaluation. Also, we release the first available LTL$_f$MUC solver for the community at https://github.com/nuutong/aaltaf-muc.git.

The rest of this paper is organized as follows. Section 2 introduces preliminaries, Section 3 presents both the theoretic foundations and implementations

of the five different approaches to compute LTL_f MUCs, Section 4 presents the experimental results, and finally, Section 5 concludes the paper.

2 Preliminaries

Linear Temporal Logic over finite traces, or LTL_f [9], extends propositional logic with finite-horizon temporal connectives. In particular, LTL_f can be considered as a variant of Linear Temporal Logic (LTL) [29]. Differently from LTL, which is interpreted over infinite traces, LTL_f is interpreted over finite traces. Given a set of atomic propositions $\mathcal{P}$, the syntax of LTL_f is identical to LTL, and defined as:

$$\varphi ::= tt \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi$$

where tt represents the *true* formula, $p \in \mathcal{P}$ is an atomic proposition, $\neg$ is the *negation*, $\wedge$ is the *and*, $\bigcirc$ is the *strong Next* and $\mathcal{U}$ is the *Until* operator. We also have the corresponding dual operators (in semantics, see below) $f\!f$ (*false*) for tt, $\vee$ (or) for $\wedge$, $\bullet$ (weak Next) for $\bigcirc$ and $\mathcal{R}$ (Release) for $\mathcal{U}$. Moreover, we use the notation $\Box\varphi$ (Global) and $\Diamond\varphi$ (Future) to represent $f\!f\,\mathcal{R}\,\varphi$ and $tt\,\mathcal{U}\,\varphi$, respectively. Notably, $\bigcirc$ is the standard *Next* operator, while $\bullet$ is *weak Next*; $\bigcirc$ requires the existence of a successor state, while $\bullet$ does not. Thus $\bullet\varphi$ is always true in the last state of a finite trace since no successor exists there.

A finite *trace* $\rho = \rho[0], \rho[1], \ldots, \rho[n]$ is a sequence of propositional interpretations (sets), in which $\rho[m] \in 2^{\mathcal{P}}$ $(0 \leq m < |\rho|)$ is the m-th interpretation of ρ, and $|\rho| = n+1$ represents the length of ρ. Intuitively, $\rho[m]$ is interpreted as the set of propositions that are *true* at instant m. We denote ρ_i to represent $\rho[i], \rho[i+1], \ldots, \rho[n]$, which is the suffix of ρ from position i.

LTL_f formulas are interpreted over finite traces. For a finite trace ρ and an LTL_f formula φ, we define the satisfaction relation $\rho \models \varphi$ (i.e., ρ is a model of φ) as follows:

- $\rho \models tt$;
- $\rho \models p$ iff $p \in \rho[0]$, where p is an atomic proposition;
- $\rho \models \neg\varphi$ iff $\rho \not\models \varphi$;
- $\rho \models \varphi_1 \wedge \varphi_2$ iff $\rho \models \varphi_1$ and $\rho \models \varphi_2$;
- $\rho \models \bigcirc\varphi$ iff $|\rho| > 1$ and $\rho_1 \models \varphi$;
- $\rho \models \bullet\varphi$ iff $|\rho| = 1$ or $\rho_1 \models \varphi$;
- $\rho \models \Diamond\varphi$ iff $\rho_i \models \varphi$ for some $0 \leq i < |\rho|$;
- $\rho \models \Box\varphi$ iff $\rho_i \models \varphi$ for every $0 \leq i < |\rho|$;
- $\rho \models \varphi_1\,\mathcal{U}\,\varphi_2$ iff there exists i with $0 \leq i < |\rho|$ such that $\rho_i \models \varphi_2$, and for every j with $0 \leq j < i$ it holds that $\rho_j \models \varphi_1$.
- $\rho \models \varphi_1\,\mathcal{R}\,\varphi_2$ iff for every $0 \leq i < |\rho|$, $\rho_i \models \varphi_2$ does not hold implies that there is $0 \leq j < i$ such that $\rho_j \models \varphi_1$.

Two LTL_f formulas φ_1 and φ_2 are semantically equivalent, denoted as $\varphi_1 \equiv \varphi_2$, iff for every finite trace ρ, $\rho \models \varphi_1$ iff $\rho \models \varphi_2$. According to the semantics of LTL_f formulas, it is trivial to have that $f\!f \equiv \neg tt$, $\bigcirc\varphi \equiv \neg\bullet\neg\varphi$, $(\varphi_1\,\mathcal{U}\,\varphi_2) \equiv$

$\neg(\neg\varphi_1 \mathcal{R} \neg\varphi_2)$ and $\Box\varphi \equiv \neg\Diamond\neg\varphi$. A *literal* is an atomic proposition $p \in \mathcal{P}$ or its negation ($\neg p$). We say an LTL_f formula is in *Negation Normal Form* (NNF) if the negation operator appears only in front of an atomic proposition. Every LTL_f formula can be converted into its NNF in linear time. **We assume that all LTL_f formulas are in NNF in this paper**, to meet the constraint of [26] that the SAT-based LTL_f satisfiability checking algorithm requires the input formula to be in NNF[5]. Given an LTL_f formula φ and its set of atomic propositions $\mathcal{P}$, the notation $2^{\mathcal{P}}$ denotes all subsets of $\mathcal{P}$. Each $A \in 2^{\mathcal{P}}$ is considered as an assignment of each atomic proposition, where a is assigned to be true if $a \in A$; Otherwise a is false. Also, we use the notation $(2^{\mathcal{P}})^+$ to represent the set of all non-empty finite traces whose each position consists of an assignment of $2^{\mathcal{P}}$.

Definition 1 (LTL_f Satisfiability). *An LTL_f formula φ is satisfiable iff there exists a (non-empty) finite trace $\rho \in (2^{\mathcal{P}})^+$ such that $\rho \models \varphi$; otherwise, it is unsatisfiable.*

Theorem 1 ([9]). *Checking the satisfiability of an LTL_f formula is PSPACE-complete.*

In this paper, we focus on the LTL_f formula with the form of $\bigwedge_{1\leq i\leq n}\varphi_i$, where φ_i is called the i-th *clause*. A clause is an LTL_f formula whose root operator is not $\wedge$. So we can represent an LTL_f formula φ by a set of clauses $\{\varphi_i \mid 1 \leq i \leq n\}$. **In the rest of the paper, we mix-use an LTL_f formula φ and its corresponding clause set.** That means, the formula φ can represent the clause set $\{\varphi_i \mid \varphi_i \textit{ is a clause of } \varphi\}$ and, a formula set $\{\varphi_i \mid \varphi_i \textit{ is a clause of } \varphi\}$ can also represent the formula $\bigwedge \varphi_i$. In particular, $\emptyset$ represents tt. Now we define the *minimal unsatisfiable core* for an LTL_f formula.

Definition 2 (Minimal Unsatisfiable Core). *A minimal unsatisfiable core (MUC) muc of a given LTL_f formula φ, is a subset of φ such that*

1. *muc is unsatisfiable;*
2. *every proper subset of muc (i.e., $\subsetneq$ muc) is satisfiable.*

By Definition 2, for a given LTL_f formula, there may be more than one subset that satisfies the above conditions, that is, one formula may have more than one MUC.

Theorem 2. *Computing an MUC for an unsatisfiable LTL_f formula is PSPACE-hard.*

Proof. According to Definition 2, the MUC can be computed in the following steps: (1) pick a clause $\varphi_i \in \varphi$ which is never chosen; (2) update $\varphi = \varphi \setminus \{\varphi_i\}$ if $\varphi \setminus \{\varphi_i\}$ is still unsatisfiable; Otherwise keep φ_i in φ; (3) repeat (1) until every clause is selected. The final φ computed from the above procedure is an MUC. Assume the number of clauses of φ is n, then this construction at most invokes n-times the checking of LTL_f satisfiability. Recall that checking LTL_f satisfiability is PSPACE-complete (Theorem 1). Therefore, the MUC computation for the unsatisfiable LTL_f formula is PSPACE-hard. $\square$

[5] The translation to the input of an SAT solver requires the input LTL_f formula to be in NNF, more details please refer to the literature [26].

3 Computing LTL_f Minimal Unsatisfiable Core

In this section, we first present two general algorithms NaiveMUC and BinaryMUC to construct an MUC for an arbitrary unsatisfiable LTL_f formula. Then we introduce heuristics based on the Boolean UC technique and integrated it into the above two general approaches to conduct NaiveMUC+UC and BinaryMUC+UC, respectively. Finally, we propose the dedicated MUC algorithm GlobalMUC for the *global* unsatisfiable formulas.

3.1 NaiveMUC and BinaryMUC: MUC Computation in General

Algorithm 1: NaiveMUC (without the part in the dashed box) and NaiveMUC+UC (with the part in the dashed box). The function implementation of `getUcFrom` is shown in Algorithm 3 with the proper explanation in Section 3.2.

```
  Input: An unsatisfiable LTL_f formula φ
  Output: An MUC for φ
1 S := clauses(φ)
2 muc := ∅
3 while S ≠ ∅ do
4     pop ψ out of S
5     if S ∧ muc is satisfiable then
6         muc := muc ∪ {ψ}
      ┌ - - - - - - - - - - - - - - - - - - - ┐
      | else                                  |
      |     S' := getUcFrom(S, muc)           |
      |     if S' ∧ muc is unsatisfiable      |
      |     then                              |
      |         S := S'                       |
7     └ - - - - - - - - - - - - - - - - - - - ┘
8 return muc
```

As mentioned in the proof of Theorem 2, a trivial way to construct an MUC is to delete one clause of the input unsatisfiable formula φ at a time and then check whether the remaining part is still unsatisfiable. If the result is unsatisfiable, the chosen clause is not in the final MUC and can be removed from the original formula. Repeating the above process for every clause in the original formula and the remaining clauses in the final formula is an MUC. This algorithm is named *NaiveMUC* and shown in Algorithm 1 (without the part inside the dashed box). The correctness of this trivial approach can be guaranteed by Theorem 3 below.

Taking the unsatisfiable LTL_f formula $\varphi = \Box\Diamond a \wedge a \wedge \Box\Diamond\neg a \wedge b$ as an example, NaiveMUC first removes the first clause from the formula to obtain the remaining formula $\varphi_1 = a \wedge \Box\Diamond\neg a \wedge b$, which is satisfiable. Therefore, the clause $\Box\Diamond a$

belongs to an MUC, and it is added to the *muc* set. Next, the second clause of φ is removed to obtain the remaining formula $\varphi_2 = \Box\Diamond a \wedge \Box\Diamond\neg a \wedge b$. Since φ_2 is unsatisfiable, the second clause a does not belong to an MUC, so it can be removed from the original formula. The simplified formula $\varphi = \Box\Diamond a \wedge \Box\Diamond\neg a \wedge b$ is obtained. This process continues by removing the third clause and performing satisfiability checking, until all clauses have been attempted to be removed once, resulting in an MUC of $\{\Box\Diamond a, \Box\Diamond\neg a\}$.

Lemma 1. *For two LTL$_f$ formulas φ_1 and φ_2 such that $\varphi_1 \subseteq \varphi_2$, φ_1 is unsatisfiable implies that φ_2 is unsatisfiable.*

Proof. We perform the proof by contradiction. We assume that φ_2 is satisfiable when φ_1 is unsatisfiable. Then there exists a finite trace ρ such that $\rho \models \varphi_2$. By $\varphi_2 = \varphi_1 \wedge (\varphi_2 \setminus \varphi_1)$, we have $\rho \models \varphi_1$, which contradicts that φ_1 is unsatisfiable. So the hypothesis does not hold, i.e., φ_2 is unsatisfiable if φ_1 is unsatisfiable. The proof is done. $\Box$

Theorem 3. *For two LTL$_f$ formulas φ_1 and φ_2 such that $\varphi_1 \subseteq \varphi_2$, φ_m is an MUC for φ_1 implies that φ_m is an MUC for φ_2.*

Proof. Since φ_m is an MUC for φ_1, φ_1 is unsatisfiable (otherwise φ_1 does not have an MUC). Also because $\varphi_1 \subseteq \varphi_2$ is true, so φ_2 is unsatisfiable according to Lemma 1. Based on Definition 2, $\varphi_m \subseteq \varphi_1$ is true, so $\varphi_m \subseteq \varphi_2$ is also true. Moreover, because φ_m is an MUC for φ_1, from Definition 2, φ_m satisfies (1) φ_m is unsatisfiable, and (2) every proper subset of φ_m is satisfiable. Therefore, φ_m is an MUC for φ_2. The proof is done. $\Box$

Obviously, the NaiveMUC approach above requires calling the LTL$_f$ satisfiability solver for each clause in the original formula. It can be improved by introducing the dichotomy strategy to reduce the invoke frequencies of the LTL$_f$-satisfiability-checking procedure that is time-consuming. The main algorithm *BinaryMUC* is shown in Algorithm 2 (without the parts inside the dashed box).

In Algorithm 2, we obtain the MUC from an LTL$_f$ formula based on the dichotomy strategy. We initialize the set S by the input formula φ (Line 1). In the *while* loop (Line 3-16), we process one element ψ of S in each round. Note that we pop ψ out of S and ψ is no longer in S (Line 4). If ψ contains only one clause (i.e., $|\psi| = 1$), this clause has to be included in *muc* (Line 5-7). Otherwise, we divide ψ into the binary partition $\langle\psi_1, \psi_2\rangle$. If $\psi_1 \wedge S \wedge muc$ is unsatisfiable, ψ_2 is discarded and ψ_1 is pushed back into S (Line 9-11). Similarly, if $\psi_2 \wedge S \wedge muc$ is unsatisfiable, ψ_1 is discarded and ψ_2 is pushed back into S (Line 12-14). If neither of the above is unsatisfiable, then both of them contain clauses in *muc*. So we put ψ_1 and ψ_2 both back into S and divided them into two parts instead of as a whole (Line 16).

Consider the formula $\varphi = \Box\Diamond a \wedge a \wedge \Box\Diamond\neg a \wedge b$, and BinaryMUC first divides the formula φ into the sub-formulas $\varphi_1 = \Box\Diamond a \wedge a$ and $\varphi_2 = \Box\Diamond\neg a \wedge b$. Then, their satisfiability is checked, and since both φ_1 and φ_2 are satisfiable, we know that they both contain part of the clauses in the MUC. Therefore, they are

both added to stack S for further processing. We then take out φ_1 from S and divide it into the sub-formulas $\varphi_{11} = \Box\Diamond a$ and $\varphi_{12} = a$. At this point, S only contains φ_2 and muc is empty, so $\varphi_{11} \wedge S \wedge muc$ is equivalent to $\varphi_{11} \wedge \varphi_2$ (i.e., $\Box\Diamond a \wedge \Box\Diamond\neg a \wedge b$). $\varphi_{11} \wedge S \wedge muc$ is unsatisfiable, so φ_{11} contains some clauses from the MUC, and since φ_{11} only contains one clause, $\Box\Diamond a$ belongs to the MUC and is added to the muc set. Since $\varphi_1 = \varphi_{11} \wedge \varphi_{12}$ is satisfiable and we already know that φ_{11} contains some clauses from the MUC, φ_{12} does not need to be checked and can be removed. We then take out φ_2 from S and repeat the process. If we rearrange the clauses in φ to obtain $\varphi' = \Box\Diamond a \wedge \Box\Diamond\neg a \wedge a \wedge b$, then after the first binary division, we can directly reduce the length of the formula by half (because $\varphi'_1 = \Box\Diamond a \wedge \Box\Diamond\neg a$ is unsatisfiable and we can simply remove the second half of the formula, $a \wedge b$).

The *Partition(ψ)* function at Line 8 divides the formula ψ into two subformulas with an equal number of clauses. And it requires ψ to contain more than two clauses. Formally, for $\psi = \{\psi_1, \cdots, \psi_n\}$ with $n \geq 2$, we have $Partition(\psi) = \langle\{\psi_1, \cdots, \psi_{\lfloor n/2 \rfloor}\}, \{\psi_{\lfloor n/2 \rfloor+1}, \cdots, \psi_n\}\rangle$. Now we prove the correctness of the algorithm.

Lemma 2. *Let $\psi = \bigwedge \psi_i$ and $\theta_k = \bigwedge_{k \neq i} \psi_i$ where $1 \leq i, k \leq |S \cup muc|$ and $\psi_i \in S \cup muc$, then we have the following properties always true in Algorithm 2:*

1. *$\psi \subseteq \varphi$ is true and ψ is unsatisfiable;*
2. *θ_k is satisfiable for every $1 \leq k \leq |S \cup muc|$.*

Proof. 1. Initially, $\psi = \varphi$ is true and ψ is unsatisfiable. In the while loop, only $\psi_2 \subseteq clauses(\varphi)$ (Line 10-11) or $\psi_1 \subseteq clauses(\varphi)$ (Line 12-13) is removed from $S \cup muc$. Therefore, ψ only consists of elements from $clauses(\varphi)$, and $\psi \subseteq \varphi$ is true. Moreover, Line 10-11 and Line 12-13 also guarantees ψ is still unsatisfiable after removing ψ_2 and ψ_1, respectively.

2. Initially, $|S| = 1$ and deleting one element makes S empty. In this case, $\theta_k = true$ and it is satisfiable. Inductively, assume θ_k is satisfiable for each k at the beginning of the while loop. After one round of the loop, there is $\varphi_i \in S$ such that only a half of $clauses(\varphi_i)$ will be kept in S (Line 10-13), or φ_i will be partitioned into two equal parts and kept in S (Line 15).
   In the former case, deleting ψ_1 or ψ_2 at Line 10-13 makes ψ satisfiable, because according to the hypothesis $S \backslash \{\varphi_i\}$ is satisfiable before the loop starts, and here $S \backslash \{\varphi_i\}$ before the loop starts equals $S \backslash \{\psi_1\}$ at Line 10-11 or $S \backslash \{\psi_2\}$ at Line 12-13. Moreover, let $\varphi_j \neq \varphi_i$ be an element in $S \backslash \{\psi_1\}$ at Line 10-11. Then θ_j is satisfiable because $S \backslash \{\psi_1, \varphi_j\}$ at Line 10-11 is a subset of $S \backslash \{\varphi_j\}$ before the loop starts. According to the hypothesis, θ_j is satisfiable before the loops start, and therefore θ_k is also satisfiable at Line 10-11. The case is similar at Line 12-13.
   In the latter case at Line 15, the formula corresponding to S does not change, since $\varphi_i = \psi_1 \cup \psi_2$ is true at Line 15. So even though φ_i is popped out from S at Line 4, S becomes the same after Line 15. It implies that $|S \cup muc|$ does not change as well. According to the hypothesis, each θ_k is satisfiable before the loop, so it is still satisfiable after Line 15. The proof is done.

Algorithm 2: BinaryMUC (without the part in the dashed box) and BinaryMUC+UC (with the part in the dashed box). The details of function `getUcFrom` are also referred to Algorithm 3 and Section 3.2.

Input: An unsatisfiable LTL_f formula φ

Output: An MUC for φ

```
1  S := {φ}
2  muc := ∅
3  while S ≠ ∅ do
4      pop ψ out of S
5      if ψ contains only one clause then
6          muc := muc ∪ ψ
7          continue
8      ⟨ψ1, ψ2⟩ := Partition(ψ)
9      if ψ1 ∧ S ∧ muc is unsatisfiable then
           ┌ let ψ1' := getUcFrom(ψ1, S ∧ muc)
           │ if ψ1' ∧ S ∧ muc is unsatisfiable
           │ then
           │     S := S ∪ {ψ1'}
10         └
11         else
12             S := S ∪ {ψ1}
13     else if ψ2 ∧ S ∧ muc is unsatisfiable then
           ┌ let ψ2' := getUcFrom(ψ2, S ∧ muc)
           │ if ψ2' ∧ S ∧ muc is unsatisfiable
           │ then
           │     S := S ∪ {ψ2'}
14         └
15         else
16             S := S ∪ {ψ2}
17     else
18         S := S ∪ {ψ1, ψ2}
19 return muc
```

□

Theorem 4. *The output of BinaryMUC in Algorithm 2 is an MUC of the input unsatisfiable formula φ.*

Proof. First Algorithm 2 can always terminate, because after every loop the size of S is reduced and it will be eventually empty. When the algorithm terminates, S is empty. From Lemma 2, the set muc satisfies (1) $muc \subseteq \varphi$ and muc is unsatisfiable, and (2) deleting every element from muc, i.e., θ_k will become satisfiable. Also in Line 5-6 of Algorithm 2, it guarantees every element of muc is a clause. Therefore, muc is exactly an MUC of φ based on Definition 2. □

3.2 NaiveMUC+UC and BinaryMUC+UC: MUC Computation using Boolean Unsatisfiable Core

The NaiveMUC and BinaryMUC approaches are logic-free to compute MUCs. For example, the algorithms can also be used to compute an MUC for an unsatisfiable Boolean formula. However, these two approaches do not utilize the inherent features dedicated to LTL$_f$, e.g., those shown in [24]. In this section, we revisit such features in LTL$_f$ and present the corresponding MUC algorithms that can leverage those heuristics.

Definition 3 (Obligation Formulas [24]). *Given an LTL$_f$ formula φ, we inductively define three kinds of obligation formulas: general obligation formula, global obligation formula and release obligation formula, denoted as $\mathsf{off}(\varphi)$, $\mathsf{ofg}(\varphi)$ and $\mathsf{ofr}(\varphi)$ respectively. (We use ofx as a generic reference to off, ofg and ofr.)*

- $\mathsf{ofx}(\varphi) = \varphi$ *if* φ *is* tt, $f\!f$ *or a literal;*
- $\mathsf{ofx}(\varphi) = \mathsf{ofx}(\varphi_1) \wedge \mathsf{ofx}(\varphi_2)$ *if* $\varphi = \varphi_1 \wedge \varphi_2$*;*
- $\mathsf{ofx}(\varphi) = \mathsf{ofx}(\varphi_1) \vee \mathsf{ofx}(\varphi_2)$ *if* $\varphi = \varphi_1 \vee \varphi_2$*;*
- $\mathsf{off}(\varphi) = \mathsf{off}(\varphi_1)$, $\mathsf{ofg}(\varphi) = f\!f$ *and* $\mathsf{ofr}(\varphi) = f\!f$ *if* $\varphi = \bigcirc\varphi_1$*;*
- $\mathsf{off}(\varphi) = \mathsf{off}(\varphi_1)$, $\mathsf{ofg}(\varphi) = tt$ *and* $\mathsf{ofr}(\varphi) = f\!f$ *if* $\varphi = \bullet\varphi_1$*;*
- $\mathsf{ofx}(\varphi) = \mathsf{ofx}(\varphi_2)$ *if* $\varphi = \varphi_1 \,\mathcal{U}\, \varphi_2$*;*
- $\mathsf{off}(\varphi) = \mathsf{ofr}(\varphi_2)$, $\mathsf{ofr}(\varphi) = \mathsf{ofr}(\varphi_2)$ *and* $\mathsf{ofg}(\varphi) = \mathsf{ofg}(\varphi_2)$ *if* $\varphi = \varphi_1 \,\mathcal{R}\, \varphi_2$*.*

Take the third item as an example, $\mathsf{ofx}(\varphi) = \mathsf{ofx}(\varphi_1) \wedge \mathsf{ofx}(\varphi_2)$ represents actually $\mathsf{off}(\varphi) = \mathsf{off}(\varphi_1) \wedge \mathsf{off}(\varphi_2)$, $\mathsf{ofr}(\varphi) = \mathsf{ofr}(\varphi_1) \wedge \mathsf{ofr}(\varphi_2)$ and $\mathsf{ofg}(\varphi) = \mathsf{ofg}(\varphi_1) \wedge \mathsf{ofg}(\varphi_2)$. In the definition, $\mathsf{ofr}(\varphi)$ is introduced to define $\mathsf{off}(\varphi)$ when the formula is a Release one. Here we list a few examples to help understand the computation process. Consider the LTL$_f$ formula $\varphi = a\,\mathcal{R}(\bullet b)$, we have $\mathsf{off}(\varphi) = \mathsf{ofr}(\bullet b) = f\!f$ and $\mathsf{ofg}(\varphi) = \mathsf{ofg}(\bullet b) = tt$. For LTL$_f$ formula $\psi = a \wedge (c\,\mathcal{U}(\bigcirc b))$, we have $\mathsf{ofr}(\psi) = \mathsf{ofr}(a) \wedge \mathsf{ofr}(c\,\mathcal{U}(\bigcirc b)) = a \wedge \mathsf{ofr}(\bigcirc b) = a \wedge f\!f$.

The following lemma shows how $\mathsf{off}()$ can help determine the satisfaction of LTL$_f$ formulas.

Lemma 3 ([24]). *For an LTL$_f$ formula φ, $\mathsf{off}(\varphi)$ is satisfiable implies φ is satisfiable.*

It should be noted that the other direction of Lemma 3 is not necessarily true. Consider the formula $\varphi = \Diamond a \wedge \Diamond\neg a$ as an example, where $\mathsf{off}(\varphi) = \mathsf{off}(\Diamond a) \wedge \mathsf{off}(\Diamond\neg a) = a \wedge \neg a = f\!f$ is unsatisfiable. Conversely, the formula φ is satisfiable. The theorem below is a direct result of Lemma 3.

Theorem 5. *For an LTL$_f$ formula φ, φ is unsatisfiable implies $\mathsf{off}(\varphi)$ is unsatisfiable.*

Motivated from Theorem 5, heuristics that are based on Boolean Unsatisfiable Core (UC) technique can be proposed to accelerate the MUC computation for unsatisfiable LTL$_f$ formulas. From the theorem we know that φ is unsatisfiable implies $\mathsf{off}(\varphi)$ is also unsatisfiable. Therefore, we can utilize the modern

SAT solvers, such as Minisat [15], to construct a UC from $\mathsf{off}(\varphi)$, which corresponds to a subset φ' of φ. If φ' is still unsatisfiable, then we can compute the MUC from φ' instead of φ, which can potentially speed up the performance.

The UC-based heuristics can be integrated into both NaiveMUC and BinaryMUC, which are shown in Algorithm 1 and 2, respectively. In the algorithms, the `getUcFrom` procedure implements the UC-based heuristics. Given $\psi_1 \wedge \psi_2$ being unsatisfiable, $\texttt{getUcFrom}(\psi_1, \psi_2)$ returns a subset ψ_1' of ψ_1 such that $\psi_1' \wedge \psi_2$ is still unsatisfiable.

Assuming $\varphi_1 = \Box\Diamond a \wedge b$ and $\varphi_2 = \Box\Diamond\neg a$, after executing `getUcFrom` (φ_1,φ_2), the formula is first converted to the $\mathsf{off}()$ form, $\mathsf{off}(\varphi_1) = \mathsf{off}(\Box\Diamond a) \wedge \mathsf{off}(b) = a \wedge b$ and $\mathsf{off}(\varphi_2) = \neg a$, and then sent to the Boolean satisfiability solver. The resulting uc is $\{a\}$, corresponding to the clause $\Box\Diamond a$ in φ_1. This method can be used to quickly reduce the length of the formula φ_1 when solving for an MUC.

To achieve this functionality, one can utilize the assumption-based SAT solver, e.g., Minisat [15], to encode ψ_1 as the set of assumptions and ψ_2 to the set of clauses. The UC ψ_1' from the SAT solver is ensured to be the subset of ψ_1.

Algorithm 3: `getUcFrom`: Implementation of the UC extractions for LTL_f formulas

Input: Two LTL_f formulas $\varphi_1 = \{\varphi_{11}, \varphi_{12}, \ldots, \varphi_{1m}\}$ and $\varphi_2 = \{\varphi_{21}, \varphi_{22}, \ldots, \varphi_{1n}\}$ such that $\varphi_1 \wedge \varphi_2$ is unsatisfiable.
Output: An LTL_f formula φ_1' such that $\varphi_1' \subseteq \varphi_1$ and $\varphi_1' \wedge \varphi_2$ is unsatisfiable.

1. $C \coloneqq \bigwedge_{1\leq i\leq m} \texttt{TCNF}(\mathsf{off}(\varphi_{1i})) \wedge \bigwedge_{1\leq j\leq n} \texttt{TCNF}(\mathsf{off}(\varphi_{2j}))$
2. Let $p_{\mathsf{off}(\psi)}$ be the Boolean variable corresponding to $\mathsf{off}(\psi)$ in $\texttt{TCNF}(\mathsf{off}(\psi))$
3. $C \coloneqq C \wedge \bigwedge_{1\leq j\leq n} p_{\mathsf{off}(\varphi_{2j})}$
4. $Assumption \coloneqq \bigwedge_{1\leq i\leq m} p_{\mathsf{off}(\varphi_{1i})}$
5. **assert** $BooleanSAT(Assumption, C) = unsat$
6. Get $uc \subseteq Assumption$ from the SAT solver
7. **return** $\varphi_1' = \{\psi \mid p_{\mathsf{off}(\psi)} \in uc\}$

The implementation of `getUcFrom` is shown in Algorithm 3. Given the two input LTL_f formulas φ_1 and φ_2, one can prepare the input for the SAT solver as follows. We first compute the *off* formula for each element in φ_1 and φ_2, based on which the *Conjunctive Normal Form* (CNF) via Tseitin transformation [36] is computed in the `TCNF` function. Generally speaking, for an arbitrary Boolean formula ψ in NNF, the Tseitin transformation computes the corresponding CNF $\texttt{TCNF}(\psi)$ in the linear cost such that ψ is satisfiable iff $p_\psi \wedge \texttt{TCNF}(\psi)$ is satisfiable where p_ψ is the Boolean variable corresponding to ψ. For details, `TCNF` is achieved inductively as follows:

1. $\texttt{TCNF}(p) = p$ and $\texttt{TCNF}(\neg p) = \neg p$;
2. $\texttt{TCNF}(\psi_1 \wedge \psi_2)$
   $= (p_{\psi_1\wedge\psi_2} \leftrightarrow p_{\psi_1} \wedge p_{\psi_2}) \wedge \texttt{TCNF}(\psi_1) \wedge \texttt{TCNF}(\psi_2)$

$= (p_{\psi_1 \wedge \psi_2} \vee \neg p_{\psi_1} \vee \neg p_{\psi_2}) \wedge (\neg p_{\psi_1 \wedge \psi_2} \vee p_{\psi_1}) \wedge (\neg p_{\psi_1 \wedge \psi_2} \vee p_{\psi_1}) \wedge \texttt{TCNF}(\psi_1) \wedge \texttt{TCNF}(\psi_2)$;

3. $\texttt{TCNF}(\psi_1 \vee \psi_2)$
$= (p_{\psi_1 \vee \psi_2} \leftrightarrow p_{\psi_1} \vee p_{\psi_2}) \wedge \texttt{TCNF}(\psi_1) \wedge \texttt{TCNF}(\psi_2)$
$= (\neg p_{\psi_1 \vee \psi_2} \vee p_{\psi_1} \vee p_{\psi_2}) \wedge (p_{\psi_1 \vee \psi_2} \vee \neg p_{\psi_1}) \wedge (p_{\psi_1 \vee \psi_2} \vee \neg p_{\psi_1}) \wedge \texttt{TCNF}(\psi_1) \wedge \texttt{TCNF}(\psi_2)$.

As a result, the input to the SAT solver has the form of $\bigwedge_{1 \le i \le m} p_{\varphi_{1_i}} \wedge \bigwedge_{1 \le j \le n} p_{\varphi_{2_j}} \wedge \bigwedge_{1 \le i \le m} \texttt{TCNF}(\varphi_{1_i}) \wedge \bigwedge_{1 \le j \le n} \texttt{TCNF}(\varphi_{2_j})$. For assumption-based SAT solvers like Minisat, they allow separating the input into two parts: the assumptions (which are essentially unit clauses) and non-assumptions (which are regular clauses). Moreover, if the input is unsatisfiable, the solvers can return a subset of assumptions that represent the *unsatisfiable core* (UC). Here, we can make $\bigwedge_{1 \le i \le m} p_{\varphi_{1_i}}$ to the assumptions (Line 5 of Algorithm 3) and the left are non-assumptions (Line 4 of Algorithm 3). After the SAT call (Line 6), the UC uc can be returned from the SAT solver and thus obtain the parts φ_1' from the original input LTL$_f$ formula φ_1 (Line 8).

The following theorem guarantees the correctness of the new algorithms integrated with the UC-based heuristics.

Theorem 6. *The outputs of NaiveMUC+UC and BinaryMUC+UC are both the MUC of the input formula.*

Proof. Compared NaiveMUC+UC to NaiveMUC in Algorithm 1, the only difference is the content in the dashed block. In this block, S' is computed by an SAT solver, which guarantees $S' \subseteq S$ holds. Moreover, S can be updated to S' only if $S' \wedge muc$ is unsatisfiable. According to Theorem 3, the MUC from S' is also the MUC of S. The proof to the correctness of BinaryMUC+UC is analogous. □

3.3 GlobalMUC: Reducing global LTL$_f$ MUC to Boolean MUC

We say an LTL$_f$ formula φ is a *global* formula if it is in the form of $\varphi = \bigwedge \varphi_i = \bigwedge \Box \psi_i$. In this section, we show that the MUC computation for an unsatisfiable global formula can be reduced to the MUC computation for an unsatisfiable Boolean formula, which is the $\mathsf{ofg}()$ formula in Definition 3.

Lemma 4 ([24]). *For a global LTL$_f$ formula $\varphi = \bigwedge \Box \psi_i$, we have that φ is satisfiable iff $\mathsf{ofg}(\varphi)$ is satisfiable.*

Based on Lemma 4, the following main theorem to guarantee the correctness of GlobalMUC is straightforward.

Theorem 7. *Given an unsatisfiable global LTL$_f$ formula $\varphi = \{\varphi_1, \cdots, \varphi_n\}$ and its obligation formula $\mathsf{ofg}(\varphi) = \{\mathsf{ofg}(\varphi_1), \cdots, \mathsf{ofg}(\varphi_n)\}$, $\{\varphi_{m_1}, \cdots, \varphi_{m_k}\}$ is an MUC for φ iff $\{\mathsf{ofg}(\varphi_{m_1}), \cdots, \mathsf{ofg}(\varphi_{m_k})\}$ is an MUC for $\mathsf{ofg}(\varphi)$, where $1 \le m_i \le n$ for $1 \le i \le k$.*

Proof. Let S_1 be the set $\{\varphi_{m_1}, \cdots, \varphi_{m_k}\}$ and S_2 be $\{\mathsf{ofg}(\varphi_{m_1}), \cdots, \mathsf{ofg}(\varphi_{m_k})\}$. It is true that $S_2 = \mathsf{ofg}(S_1)$. Also let S_1' be a proper subset of S_1 and S_2' be the corresponding subset of S_2 such that $S_2' = \mathsf{ofg}(S_1')$. According to Definition 2, S_2 is an MUC for $\mathsf{ofg}(\varphi)$ iff (1) S_2 is unsatisfiable, and (2) every proper subset S_2' of S_2 is satisfiable. From Lemma 4, S_1 is unsatisfiable iff S_2 is unsatisfiable. Also, every proper subset S_1' of S_1 is satisfiable iff every proper subset S_2' of S_2 is satisfiable, based on Lemma 4. So S_1 is an MUC for φ iff S_2 is an MUC for $\mathsf{ofg}(\varphi)$. □

Theorem 7 guarantees that, to compute an MUC for $\varphi = \bigwedge \Box\psi_i$, we can first compute the MUC for $\mathsf{ofg}(\varphi)$ by leveraging the state-of-the-art Boolean MUC solvers, such as MUser2 [3]. Notably, the preparation of the input to MUC solvers is analogous to that of the input to SAT solvers, so details are omitted here. After that, we can locate the subset φ' of φ which corresponds to the Boolean MUC based on Theorem 7. It should be highlighted that there is only one Boolean MUC call necessary to compute an MUC for an unsatisfiable global LTL_f formula. Therefore, this dedicated algorithm can be much faster than the general ones presented in previous sections.

4 Experimental Evaluation

4.1 Experimental Set-up

Table 1: Average solution time (millisecond) on unsatisfiable formulas

Formula type	Number of formulas	NaiveMUC	NaiveMUC +UC	BinaryMUC	BinaryMUC +UC
/acacia/demo-v3	11	174	1	10	0
/alaska/lift	131	85307	56885	16025	13303
/anzu/amba	20	244519	238289	1284	1197
/anzu/genbuf	20	241733	232900	12563	16948
/forobots	38	20	1	11	0
/rozier/counter	76	78974	20109	407	239
/schuppan/O1formula	27	64117	13	47	15
/schuppan/O2formula	27	90820	97775	92757	96070
/schuppan/phltl	49	311726	4213	16297	6236
/trp/N12x	419	48168	48160	47976	47977
/trp/N5x	250	73	88	23	21
/unsat	224	1	4	1	4
Total	1292	1165632	698438	187401	182010

Tools. We implemented the five approaches in the tool aaltaf-muc, using aaltaf [23] as the LTL_f satisfiability checker. For the GlobalMUC approach, we use MUser2 [3] as the Boolean MUC solver. Also, to ensure the correctness of the

implementation, we integrated the functionality to check whether the outputs of the five approaches are really an MUC of the input formula, which is enabled by the "-c" flag in aaltaf-muc. The principle behind is as follows: (1) first check whether the output formula set is still unsatisfiable, and (2) delete each element of the formula set to check whether the left part is satisfiable. The output is an MUC iff it can pass both two checks above. The whole tool is implemented in C++ and can be run on a Linux system with GCC version greater than 4.7.0.

Because of the correlation between the MUC and UC, we consider the UC computation approaches proposed in [30]. In the literature, four different approaches were presented to compute the Unsatisfiable Core (UC) for LTL_f. Among them, three were motivated by computing LTL UCs introduced in [7, 31] and the other was by our previous work on LTL_f satisfiability checking [23]. Notably, both the results from [30] and our preliminary experiments show that the UC computation approach based on our previous work, which is named AALTAF-UC here, computes UCs faster than the other three in most of the tested cases. In about half of the tested cases, AALTAF-UC returns the smallest UCs as well. Therefore, we only involve the comparison to AALTAF-UC in this paper.

Benchmarks. Since MUC can be computed only for unsatisfiable formulas, we select the unsatisfiable formulas from the widely-used LTL_f benchmarks in [23]. Also, due to the fact that LTL and LTL_f formulas have the same syntax and an LTL formula φ being unsatisfiable implies the LTL_f formula φ is unsatisfiable as well [23], we select the LTL benchmarks for debugging from [27, 12]. In these benchmarks, general formulas have the form $\bigwedge_i \varphi_i$, and global formulas have the form $\Box(\bigwedge_i \varphi_i)$. In total, there are 1292 unsatisfiable LTL_f formulas and 665 global ones. The different types of benchmarks are shown in the first column of Table 1.

Platform. We ran the experiments on a RedHat 6.0 cluster with 2304 processor cores in 192 nodes (12 processor cores per node), running at 2.83 GHz with 48GB of RAM per node. Each tool was executed on a dedicated node with a timeout of five minutes, measuring execution time with the *time* command. The timeout instances will get a 300-second runtime as a penalty[6]. Excluding timeouts, all outputs from the five approaches pass the MUC check (using the "-c" flag in aaltaf-muc) successfully.

4.2 Comparison among different MUC-computation approaches

Figure 1 shows that BinaryMUC has a significant performance improvement when compared with NaiveMUC. The time unit in the figure is mill-seconds and the same applies to others. Within the 5-minute timeout, the BinaryMUC can solve 80 more formulas than NaiveMUC. The average solution time of BinaryMUC is roughly 20 seconds, which is two times as fast as NaiveMUC. The reason

[6] In the scatter plots (Fig. 1-4 and Fig. 9(a)), timeout instances are plotted on the axis boundaries.

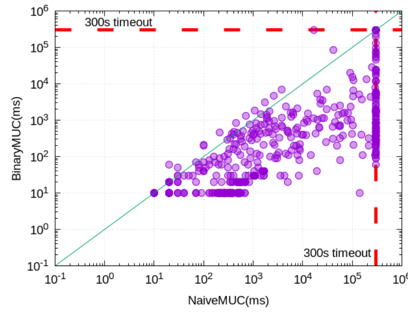

Fig. 1: Comparison between NaiveMUC and BinaryMUC on general formulas.

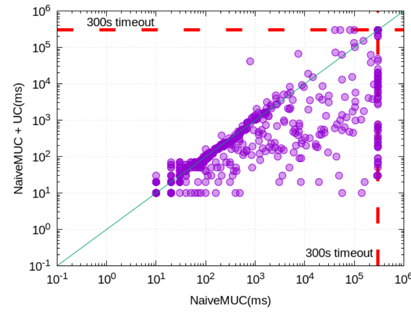

Fig. 2: Comparison between NaiveMUC and NaiveMUC+UC on general formulas.

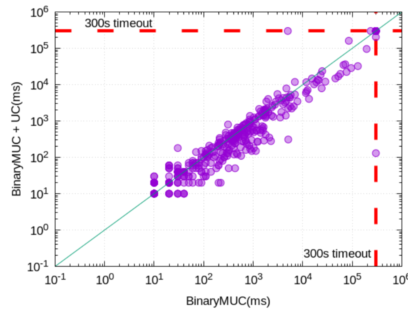

Fig. 3: Comparison between BinaryMUC and BinaryMUC+UC on general formulas.

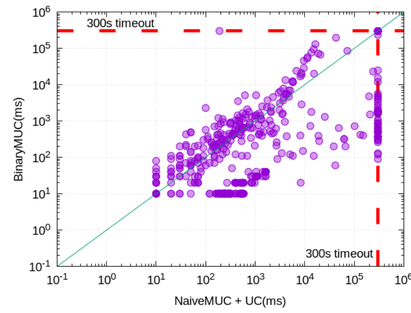

Fig. 4: Comparison between NaiveMUC+UC and BinaryMUC on general formulas.

why BinaryMUC performs well is that the dichotomy strategy can quickly reduce the length of the formula. In the best case, the length of the formula can be reduced in half at one time, thereby increasing the speed of computation.

Figure 2 shows that NaiveMUC+UC performs better than NaiveMUC. Compared to NaiveMUC, NaiveMUC+UC can solve 35 more formulas. Among all the tested instances, the number of formulas that NaiveMUC+UC solves faster is more than twice than that of NaiveMUC solves faster. In the cases when NaiveMUC+UC performs better, the average speed is 27 seconds faster than that of NaiveMUC. But when NaiveMUC performs better, the average speed is only 9 seconds faster than that of NaiveMUC+UC. The observation is that, when UC is effective, the computation speed can be significantly improved. We further analyze the reasons why using UC do not always perform better as follows. In fact, computing UC needs additional time cost, and the worse case is computing

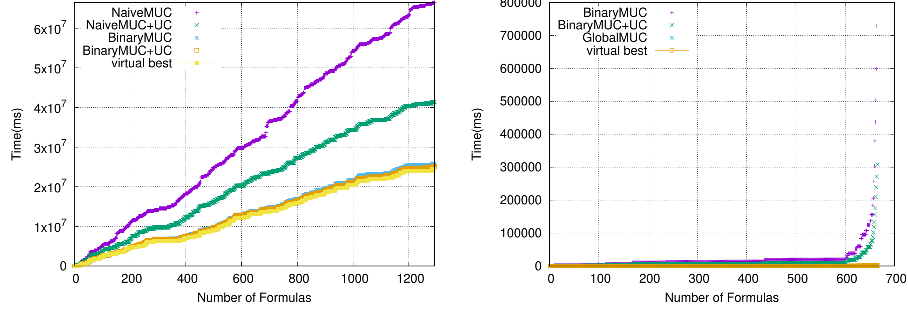

Fig. 5: Comparison among NaiveMUC, NaiveMUC+UC, BinaryMUC and BinaryMUC+UC on general formulas.

Fig. 6: Comparison among BinaryMUC, BinaryMUC+UC and GlobalMUC on $\square$- formulas.

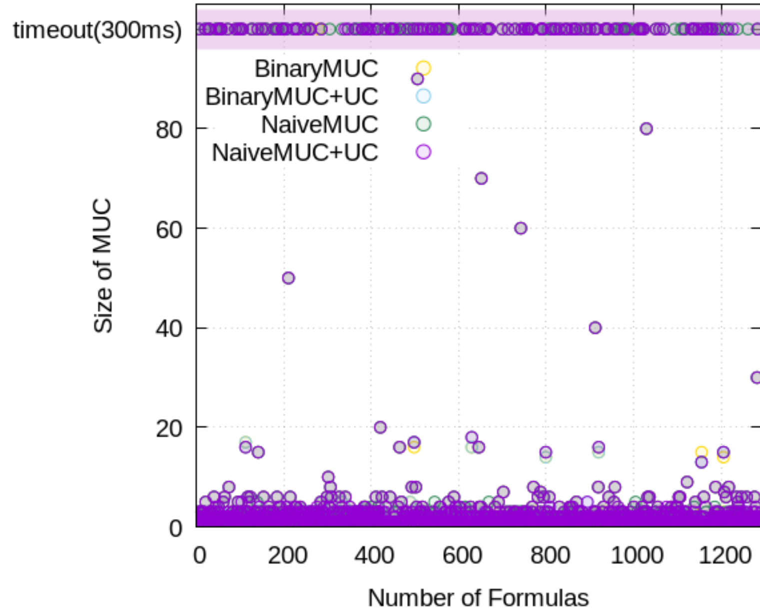

Fig. 7: Comparison on the sizes of computed MUCs from four different approaches.

UC can only waste time, e.g., the LTL_f formulas corresponding to the UC are satisfiable.

Figure 3 shows that BinaryMUC+UC performs better than BinaryMUC in the overall performance. Explicitly, there are 227 formulas that can be solved faster by using BinaryMUC+UC and 178 formulas can be solved faster by using BinaryMUC. The reason why using UC does not speed up the computation for certain instances, is the same as above.

Figure 4 shows the results between NaiveMUC+UC and BinaryMUC. It can be seen that BinaryMUC has better performance. For details, BinaryMUC can solve 45 more formulas than NaiveMUC+UC and has an average solution time of about 20 seconds. Meanwhile, the average solution time of NaiveMUC+UC is about 33 seconds, which is 13 seconds slower.

Figure 5 shows the cumulative solution time of the four algorithms on general formulas. From the figure, we can see that the BinaryMUC algorithm achieves a >50% speed-up over the NaiveMUC algorithm, while BinaryMUC+UC has only a small advantage over BinaryMUC, which is consistent with the results discussed earlier. Moreover, the *virtual best* results among different approaches are plotted in the figure, and BinaryMUC+UC performs almost as well as the virtual best one. Table 1 lists the average solution time of these four algorithms on different types of general formulas. We can find that the solution speed of the algorithm is related to the specific structure of the formula, but in general, using UC makes the algorithm perform better.

According to the analysis of the above five figures, we conclude that for a general LTL_f formula, BinaryMUC+UC performs best, followed by BinaryMUC, NaiveMUC+UC, and NaiveMUC in order. For the general approaches NaiveMUC and BinaryMUC, better performance can be obtained after integrating with the UC heuristic.

Figure 6 shows that GlobalMUC performs more than 300 times faster than BinaryMUC+UC and BinaryMUC. The evaluations on NaiveMUC and NaiveMUC+UC are not considered here as they are not competitive in previous experiments. Even more, GlobalMUC performs the same as the virtual best one which collects all the best results from different approaches. Based on Theorem 4, the GlobalMUC approach only needs to call the MUser2 tool once after converting the LTL_f formula into its $\mathsf{ofg}()$ Boolean formula, which significantly improves the performance. For global formulas, GlobalMUC has the best performance, gaining a 300× speed-up when compared to the other two competitive solutions.

Figure 7 shows the comparison of the sizes of computed MUCs from the four different approaches[7]. For nearly 98% of all the instances excluding those timeout, every approach computes an MUC with a size smaller than 10. The figure shows clearly that for a given instance, the sizes of MUCs computed by different approaches do not vary significantly.

We also use the Jaccard index to measure the overlap of computed MUCs among different approaches. Firstly, when comparing BinaryMUC with Binary-

[7] Timeout instances are plotted on the bounds of the y-axis with the MUC size being 100.

MUC+UC, the Jaccard index formula we use to measure the overlap is $(|M_1 \cap M_2|)/(|M_1 \cup M_2|)$, where M_1, M_2 are the MUCs computed by BinaryMUC and BinaryMUC+UC respectively. The Jaccard index value for each instance is plotted in Figure 8(a). It turns out that BinaryMUC and BinaryMUC+UC compute the same MUCs (the Jaccard index is 1) on more than 90% of the instances (1219 out of 1291) solved by both approaches, and the same situation occurs with NaiveMUC and NaiveMUC+UC. Introducing the UC heuristics seems mainly to speed up the MUC computation. However, as shown in Figure 8(b), when considering BinaryMUC+UC and NaiveMUC+UC, the ratio of the same output MUCs reduced to 46% (589 out of 1291), which is reasonable, because they use different strategies to enumerate elements of the original formula, thus constructing different MUCs. Finally, it is worth noting that, there are 44 (resp. 642) out of 1291 instances for that BinaryMUC and BinaryMUC+UC (resp. BinaryMUC+UC and NaiveMUC+UC) compute completely different MUCs without any overlap (Jaccard index is 0).

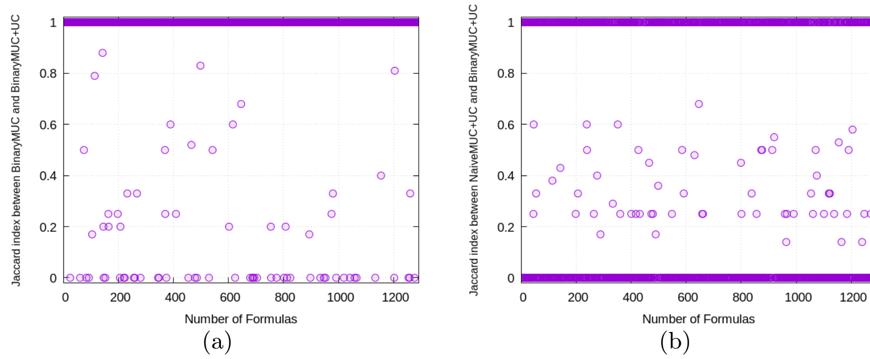

Fig. 8: Plots on the Jaccard index between BinaryMUC vs. BinaryMUC+UC (left) and NaiveMUC+UC vs. BinaryMUC+UC (right).

In summary, we conclude from the experimental results that 1) BinaryMUC is better than NaiveMUC; 2) Applying the UC-based heuristics to BinaryMUC is able to improve the performance with an approximate 10% speed-up, and 3) GlobalMUC is the best solution to compute MUCs for global LTL_f formulas.

4.3 Comparison to UC-computation approaches in [30]

We compare the best MUC-computation approach, i.e., BinaryMUC+UC, to the best UC-computation approach, i.e., AALTAF-UC. Firstly, we compare the time cost of these two approaches for each instance, which is shown in Fig. 9(a). In terms of time-consuming for each instance, AALTAF-UC can outperform BinaryMUC+UC on 74% of the benchmarks, while BinaryMUC+UC is able to outperform AALTAF-UC on 25% of the benchmarks. It is reasonable that

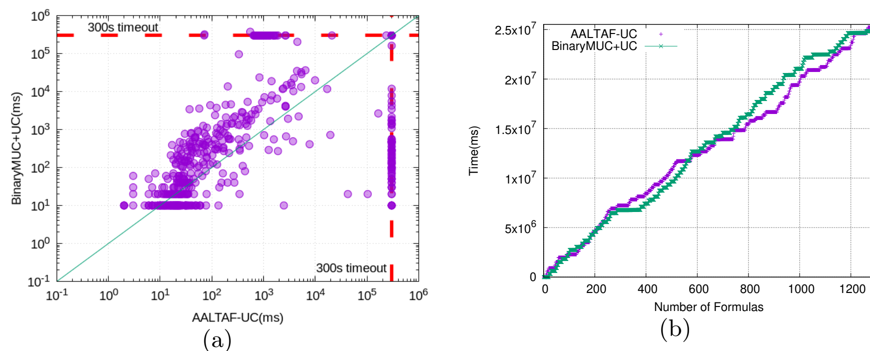

Fig. 9: Comparison between AALTAF-UC and BinaryMUC+UC on general formulas.

AALTAF-UC can be faster than BinaryMUC+UC because computing UCs is a simpler task than computing MUCs. However, if we consider the accumulated time that is shown in Fig. 9(b), the total time costs of BinaryMUC+UC and AALTAF-UC are nearly the same. There are 81 (resp. 78) instances in total for which AALTAF-UC (resp. BinaryMUC+UC) cannot compute the UC (resp. MUC) (see those plots on the bound of axis in Fig. 9(a)). There are two more timeout cases solved by AALTAF-UC than BinaryMUC+UC, thus yielding a 900-seconds more accumulated penalty time. Therefore, although AALTAF-UC can outperform BinaryMUC+UC in more cases, the total accumulated times for both approaches are similar.

Secondly, the results show that the sizes of computed MUCs are in general smaller than those of computed UCs, see Fig. 10(a).[8] It is obvious that for most of the instances, the MUCs computed from our best approach BinaryMUC+UC have smaller sizes than that of UCs computed by AALTAF-UC. However, other approaches in [30] may produce smaller UCs than AALTAF-UC, and our results are not sufficient to show the BinaryMUC+UC can have smaller MUC sizes than any algorithm presented in the literature [30]. Please note that some of the samples in the figure have relatively large MUC sizes, even exceeding 60. These samples are from the *schuppan/O2formula* benchmark, and for these cases, the UC-extraction algorithms cannot even return a result. Therefore, we show that computing MUCs by BinaryMUC+UC does not cost significant overhead when compared to the UC computation by AALTAF-UC. Also towards the overlap comparison between BinaryMUC+UC and AALTAF-UC, the plots are shown in Figure 10(b). AALTAF-UC and BinaryMUC+UC compute the same UCs on 729 out of 1291 instances (the ratio is 56%) and completely different UCs on 143 out of 1291 cases (the ratio is 11%) without any overlap. The results show that about half of the UCs computed by AALTAF-UC are already minimal, which

[8] Timeout instances are plotted on the bounds of the y-axis with an MUC (UC) size equals to 100.

is consistent with the observation from [30] (in these cases AALTAF-UC can compute smaller sizes of UCs than other approaches). It is surprising to see that computing UCs via the conflict sequence, which is presented in our previous work [26], are MUCs. The reason will be explored in our future work.

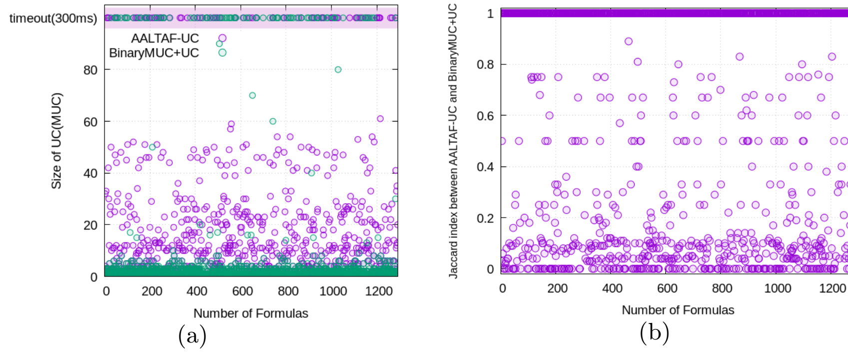

Fig. 10: (a) Comparison on the sizes of computed MUCs (UCs) between AALTAF-UC and BinaryMUC+UC. (b) Plots on the Jaccard index of AALTAF-UC vs. BinaryMUC+UC.

5 Concluding Remarks

In this paper, we focus on the MUC problem of unsatisfiable LTL_f formulas and present five different solutions, including two generic and three dedicated ones for LTL_f. We then fully explore the performance among these approaches by an extensive experimental evaluation and show that the GlobalMUC is the best approach to compute MUCs for global formulas, while BinaryMUC+UC is the best option to compute MUCs for an arbitrary unsatisfiable LTL_f formula. We implement these approaches into our tool aaltaf-muc, and to the best of our knowledge, aaltaf-muc is the only available solver that provides MUC computation for LTL_f. We also compared to the latest work on LTL_f UC computation, and show that computing the MUC can still have significant advantages.

In the current stage, the satisfiability solver is used as a black box to compute MUCs for unsatisfiable LTL_f formulas, whose knowledge of unsatisfiability is discarded. In the future, we consider extracting the UC directly from the satisfiable solver to help accelerate the MUC computation. Also, LTL_f has been widely used in AI-related applications such as planning, and we will investigate proper scenarios in the real world to apply our MUC computation methods to accelerate the necessary inconsistency checking for specifications written in LTL_f.

Acknowledgment. We thank anonymous reviewers for their helpful comments. This work is supported by the National Natural Science Foundation of China (Grant NO. U21B2015 and 62002118), the Shanghai Collaborative Innovation Center of Trusted Industry Internet Software, and the National Key Research and Development Program (Grant NO. 2022YFB3305200)

References

1. Bacchus, F., Kabanza, F.: Planning for temporally extended goals. Ann. of Mathematics and Artificial Intelligence **22**, 5–27 (1998)
2. Bansal, S., Li, Y., Tabajara, L., Vardi, M.: Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence. pp. 9766–9774. AAAI Press (2020)
3. Belov, A., Marques-Silva, J.: Muser2: An efficient mus extractor. Journal on Satisfiability, Boolean Modeling and Computation **8**, 123–128 (2012)
4. Bryant, R.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys **24**(3), 293–318 (1992)
5. Calvanese, D., De Giacomo, G., Vardi, M.: Reasoning about actions and planning in LTL action theories. In: Principles of Knowledge Representation and Reasoning. pp. 593–602. Morgan Kaufmann (2002)
6. Camacho, A., Baier, J., Muise, C., McIlraith, A.: Bridging the gap between LTL synthesis and automated planning. Tech. rep., U. Toronto (2017), http://www.cs.toronto.edu/ acamacho/papers/camacho-genplan17.pdf
7. Cimatti, A., Roveri, M., Schuppan, V., Tonetta, S.: Boolean abstraction for temporal logic satisfiability. In: Proc. 15th Int'l Conf. on Computer Aided Verification. Lecture Notes in Computer Science, vol. 4590, pp. 532–546. Springer (2007)
8. De Giacomo, G., Masellis, R.D., Montali, M.: Reasoning on LTL on finite traces: Insensitivity to infiniteness. In: AAAI. pp. 1027–1033 (2014)
9. De Giacomo, G., Vardi, M.: Linear temporal logic and linear dynamic logic on finite traces. In: IJCAI. pp. 2000–2007. AAAI Press (2013)
10. De Giacomo, G., Vardi, M.: Automata-theoretic approach to planning for temporally extended goals. In: Proc. European Conf. on Planning. pp. 226–238. Lecture Notes in AI 1809, Springer (1999)
11. De Giacomo, G., Favorito, M.: Compositional approach to translate ltlf/ldlf into deterministic finite automata. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS). vol. 14 (2021)
12. Degiovanni, R., Molina, F., Regis, G., Aguirre, N.: A genetic algorithm for goal-conflict identification. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 520–531 (2018)
13. Degiovanni, R., Ricci, N., Alrajeh, D., Castro, P., Aguirre, N.: Goal-conflict detection based on temporal satisfiability checking. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 507–518. IEEE (2016)
14. Di Ciccio, C., Maggi, F.M., Montali, M., Mendling, J.: Resolving inconsistencies and redundancies in declarative process models. Information Systems **64**, 425–446 (2017)
15. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT. pp. 502–518 (2003)
16. Felfernig, A., Friedrich, G.E., Jannach, D., Stumptner, M.: Consistency-based diagnosis of configuration knowledge bases. In: Proceedings of the 14th European Conference on Artificial Intelligence. p. 146–150. ECAI'00, IOS Press, NLD (2000)

17. Fisher, M., Dixon, C., Peim, M.: Clausal temporal resolution. ACM Trans. Comput. Log. **2**(1), 12–56 (2001)
18. Giacomo, G.D., Favorito, M., Li, J., Vardi, M.Y., Xiao, S., Zhu, S.: Ltlf synthesis as and-or graph search: Knowledge compilation at work. In: Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence. pp. 3292–3298. AAAI Press (2022)
19. Goré, R., Huang, J., Sergeant, T., Thomson, J.: Finding minimal unsatisfiable subsets in linear temporal logic using bdds. https://www.timsergeant.com/files/pltlmup/gore_huang_sergeant_thomson_mus_pltl.pdf (2013)
20. Hantry, F., Saïs, L., Hacid, M.: On the complexity of computing minimal unsatisfiable LTL formulas. CoRR **abs/1203.3706** (2012), http://arxiv.org/abs/1203.3706
21. Hustadt, U., Konev, B.: Trp++ 2.0: A temporal resolution prover. In: In Proc. CADE-19, LNAI. pp. 274–278. Springer (2003)
22. Junker, U.: Quickxplain: Preferred explanations and relaxations for over-constrained problems. In: AAAI Conference on Artificial Intelligence. p. 167–172. AAAI'04, AAAI Press (2004)
23. Li, J., Rozier, K.Y., Pu, G., Zhang, Y., Vardi, M.Y.: Sat-based explicit ltlf satisfiability checking. In: The Thirty-Third AAAI Conference on Artificial Intelligence. pp. 2946–2953. AAAI Press (2019)
24. Li, J., Zhang, L., Pu, G., Vardi, M.Y., He, J.: LTL_f satisfibility checking. In: ECAI. pp. 91–98 (2014)
25. Li, J., Zhu, S., Pu, G., Vardi, M.: SAT-based explicit LTL reasoning. In: HVC. pp. 209–224. Springer (2015)
26. Li, J., Zhu, S., Pu, G., Zhang, L., Vardi, M.Y.: Sat-based explicit ltl reasoning and its application to satisfiability checking. Formal Methods in System Design pp. 1–27 (2019)
27. Luo, W., Wan, H., Song, X., Yang, B., Zhong, H., Chen, Y.: How to identify boundary conditions with contrasty metric? In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 1473–1484. IEEE (2021)
28. Patrizi, F., Lipoveztky, N., De Giacomo, G., Geffner, H.: Computing infinite plans for LTL goals using a classical planner. In: IJCAI. pp. 2003–2008. AAAI Press (2011)
29. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). pp. 46–57 (Oct 1977). https://doi.org/10.1109/SFCS.1977.32
30. Roveri, M., Di Ciccio, C., Di Francescomarino, C., Ghidini, C.: Computing unsatisfiable cores for ltlf specifications (2022). https://doi.org/10.48550/ARXIV.2203.04834, https://arxiv.org/abs/2203.04834
31. Schuppan, V.: Towards a notion of unsatisfiable and unrealizable cores for ltl. Sci. Comput. Program. **77**(7–8), 908–939 (Jul 2012). https://doi.org/10.1016/j.scico.2010.11.004
32. Schuppan, V.: Extracting unsatisfiable cores for ltl via temporal resolution. In: 2013 20th International Symposium on Temporal Representation and Reasoning. pp. 54–61 (2013). https://doi.org/10.1109/TIME.2013.15
33. Shi, Y., Xiao, S., Li, J., Guo, J., Pu, G.: Sat-based automata construction for ltl over finite traces. In: 2020 27th Asia-Pacific Software Engineering Conference (APSEC). pp. 1–10 (2020). https://doi.org/10.1109/APSEC51365.2020.00008
34. Tabajara, L., Vardi, M.: Partitioning techniques in ltlf synthesis. In: Proceedings of the 28th International Joint Conference on Artificial Intelligence. pp. 5599–5606. IJCAI 19, AAAI Press (2019)

35. Torlak, E., Chang, F.S.H., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008: Formal Methods. pp. 326–341. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
36. Tseitin, G.S.: On the Complexity of Derivation in Propositional Calculus, pp. 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg (1983)
37. Van Lamsweerde, A., Darimont, R., Letier, E.: Managing conflicts in goal-driven requirements engineering. IEEE transactions on Software engineering **24**(11), 908–926 (1998)
38. Xiao, S., Li, J., Zhu, S., Shi, Y., Pu, G., Vardi, M.Y.: On the fly synthesis for ltl over finite traces. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2021, February 7-12, 2021. pp. 6530–6537. AAAI Press (2021)
39. Zhu, S., Giacomo, G.D., Pu, G., Vardi, M.Y.: Ltl$f$ synthesis with fairness and stability assumptions. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, New York, NY, USA, February 7-12, 2020. pp. 3088–3095. AAAI Press (2020)
40. Zhu, S., Pu, G., Vardi, M.Y.: First-order vs. second-order encodings for ltlf-to-automata translation. In: Theory and Applications of Models of Computation - 15th Annual Conference, TAMC 2019, Kitakyushu, Japan, April 13-16, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11436, pp. 684–705 (2019)