The Journal of Systems and Software 213 (2024) 112036

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Goal-conflict identification based on local search and fast boundary-condition verification based on incremental satisfiability filter☆

Weilin Luo [a], Polong Chen [a], Hai Wan [a,*], Hongzhen Zhong [a], Shaowei Cai [b,*], Zhanhao Xiao [c]

[a] School of Computer Science and Engineering, Sun Yat-sen University, China
[b] Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China
[c] School of Computer Science, Guangdong Polytechnic Normal University, China

ARTICLE INFO

Keywords:
Goal-conflict identification
Boundary condition
Local search
LTL satisfiability checking

ABSTRACT

Identifying *boundary conditions (BCs)* is of fundamental importance for goal-conflict analysis. BCs are able to capture particular combinations of circumstances that make some special conflicts, namely *goal divergences*, in which the goals of the requirement cannot be satisfied as a whole. Unfortunately, existing approaches only handle small-scale cases within a reasonable running time and do not consider how to search *more general* BCs because of the search space explosion and the high computational cost of verifying BCs.

In this paper, we observed that the BCs identified by the state-of-the-art (SOTA) approach share similarities in structure. It inspires us to develop a BC identification approach, named LOGION, based on local search, to better exploit the similarity in structures. Specifically, we utilize the structural similarity of BCs to construct the neighborhood relation of our local search algorithm. Besides, we design strengthening and weakening operators of formulae to capture the semantics of more general BCs in local searching. In order to boost the BC verification, we propose a *lasso-based incremental satisfiability filter (LISF)*. The primary intuition behind LISF is using trace checking as a light-weight pre-checking, and if it is not successful, to call the satisfiability checking. We conduct extensive experiments to evaluate LOGION, comparing against SOTA approaches. Experimental results show that LOGION produces about 1 order of magnitude more general BCs than the SOTA approaches. It is also shown that LISF significantly boosts the speed, especially for large-scale cases (up to about 1 order of magnitude faster on average).

1. Introduction

In software development, *Goal-oriented requirements engineering (GORE)* (Van Lamsweerde, 2009) drives the requirements process through high-level goals, where requirement specifications are compiled into *domain properties* and *goals* to characterize the problem world and the behavior of a system, respectively. Formal requirements engineering methodologies usually use some formal languages, *e.g.*, *linear temporal logic (LTL)*, to accurately express specifications so that efficient (semi-)automatic reasoning can be performed. Numerous studies (Alrajeh et al., 2009; Degiovanni et al., 2014; Ellen et al., 2014) have highlighted the considerable benefits that formal and goal-oriented methodologies offer in producing correct specifications. Besides, GORE methodologies are widely used in other fields such as data warehouses (Maté et al., 2014) and cloud migration (Fahmideh and Beydoun, 2018).

Goal-conflict analysis has been widely used in GORE, which aims to identify, assess, and resolve conflicts that may obstruct the satisfaction of the goals. Conflicts in requirements are complex and varied. For example, checking requirements satisfiability corresponds to determining whether there are contradictions in the requirements. Such an analysis, however, is able to only find the simplest kinds of conflict in requirements. In many cases, requirements as a whole are satisfied, but the satisfaction of some goals inhibits the satisfaction of other goals. This special conflict is *goal divergences* (divergences for short), which we focus on in this paper. Identifying divergences early in the development process is important because it enables engineers to improve specifications and gain greater insight into the root causes of potential goals not being met.

Divergences can be captured by *boundary conditions (BCs)* (Definition 1). Intuitively, a BC captures a particular combination of circumstances where the goals cannot be satisfied as a whole. As a matter

☆ Editor: Xiao Liu.

* Corresponding authors.
E-mail addresses: luowlin5@mail.sysu.edu.cn (W. Luo), chenplong@mail2.sysu.edu.cn (P. Chen), wanhai@mail.sysu.edu.cn (H. Wan), zhonghzh5@mail2.sysu.edu.cn (H. Zhong), caisw@ios.ac.cn (S. Cai), xiaozhanhao@gpnu.edu.cn (Z. Xiao).

https://doi.org/10.1016/j.jss.2024.112036
Received 3 October 2022; Received in revised form 23 December 2023; Accepted 22 March 2024
Available online 3 April 2024

of fact, various approaches (Van Lamsweerde et al., 1998; Degiovanni et al., 2016, 2018b) have been proposed in order to automatically identify BCs in the context of GORE. However, as the number of identified BCs increases (there are more than 100 BCs in the case named London Ambulance Service (LAS) in Degiovanni et al. (2018b)), the assessment and resolution steps become very expensive and even impractical. To provide engineers with an acceptable number of BCs to be analyzed, the *more general (or weaker) BC* (Definition 2) has been proposed. Intuitively, the more general BC has a higher quality because it covers more combinations of circumstances of divergences. Once it is resolved, less general BCs are also resolved. We use Example 1 to illustrate the above concepts.

Example 1 (*MinePump Kramer et al., 1983*)**.** Consider a system to control a pump (p) inside a mine. The system has two sensors. One detects the high water level (h), the other detects methane in the environment (m). Domain property and goals are represented via the following LTL formulae.

Domain Property:

1. **Name**: PumpEffect
   Description: The pump is turned on for two time steps, then in the following one the water level is not high.
   Formula: $\Box((p \wedge \bigcirc p) \rightarrow \bigcirc(\bigcirc\neg h))$

Goals:

1. **Name**: NoFlooding
   Description: When the water level is high, the system should turn on the pump.
   Formula: $\Box(h \rightarrow \bigcirc(p))$
2. **Name**: NoExplosion
   Description: When there is methane in the environment, the pump should be turned off.
   Formula: $\Box(m \rightarrow \bigcirc(\neg p))$

Although the specification is consistent, *i.e.*, all domain properties and goals can simultaneously be satisfied, this specification can exhibit some divergences. One of the BCs is $\phi = \Box(h \wedge m)$ which captures the situation where the high water level and the methane always occurs. In this situation, it is not possible to satisfy both goals at the same time given the constraints of the domain property. However, a more general BC $\varphi = \Diamond(h \wedge m)$ captures more circumstances where the high water level and the methane occur at some point in the system life time.

There are two challenges to identify BCs. First, since a BC has only semantic constraints but no syntactic constraints, there is a search space explosion problem. Besides, the computational cost of verifying BCs is high because the verification requires multiple calls of an LTL satisfiability checking, which is PSPACE-complete (Sistla and Clarke, 1985). Previous approaches to identify BCs can be mainly categorized into the construct-based approaches and the search-based approaches. Construct-based approaches focus on generating BCs based on constructing templates (Van Lamsweerde et al., 1998) or tableau structures (Degiovanni et al., 2016). However, the former is less general because the templates cannot be applied in all cases, and the latter suffers from performance and scalability issues resulting from its sophisticated logical mechanisms. Recently, a search-based approach based on genetic algorithm (Degiovanni et al., 2018b) to find BCs has been proposed to improve the performance. It maintains a population representing the candidate BCs and generates BCs as descendants via crossover and mutation operations as much as possible. Unfortunately, there is no work to propose an approach for identifying more general BCs. The work (Degiovanni et al., 2018b) introduced the definition of the general BC, but their algorithm does not produce bias to search for more general BCs. In a nutshell, previous approaches only handle specifications with a limited scale within a reasonable running time and cannot guide the identification of more general BCs.

In this paper, we quantitatively analyze the reasons why the state-of-the-art (SOTA) approach (Degiovanni et al., 2018b) has achieved great performance improvements. Based on the tree edit distance (Zhang and Shasha, 1989), we propose the formula distance to measure the syntactic similarity of the BCs. We observe that some pairs of BCs are similar in structure, which occurs frequently in cases. Recalling that ϕ and φ presented in Example 1, they are similar in structure: they differ from each other in only one place ($\Box$ and $\Diamond$). Consequently, once one BC is identified, the other, even more general one, may be found by simply modifying the identified BC. A reasonable hypothesis is that the performance improvement of the SOTA approach comes from its identification of similar BCs, which can serve as a heuristic for efficiently identifying BCs.

Motivated by the observation, to better exploit structural similarity, we propose a novel local search algorithm, called LOGION, which combines the syntax and semantics of the LTL formula to identify more general BCs. Specifically, LOGION starts with a candidate BC and searches for its neighbors, syntactically similar formulae to the candidate BC, by syntactically mutating it. Then, it selects a neighbor as a new candidate BC and starts a new iteration. The core of our approach is the *strengthening* and *weakening* operators for an LTL formula based on syntax, which constructs the semantics connection between formulae. LOGION performs the strengthening or weakening operator on a candidate solution according to the result of the BC checking to obtain a new candidate solution.

To speed up the BC verification, we propose a *lasso-based incremental LTL satisfiability filter (LISF)* to reduce the calls to the LTL satisfiability solver. A lasso, a linear-time structure with a loop, is a single trace. Checking whether a lasso is a model of an LTL formula is in bilinear time (Markey and Schnoebelen, 2003), which is much faster than checking LTL satisfiability that is PSPACE-complete (Sistla and Clarke, 1985). The underlying idea of LISF is first to apply a fast but incomplete checking for lassos, and if not successful, to invoke a satisfiability solver.

We assess LOGION on the artificial cases introduced by Degiovanni et al. (2018b).[1] To evaluate the practicality, we add cases sourcing from the industry. Compared with the artificial cases, the industrial case is more challenging since it maintains a larger scale specification. Experimental results show that LOGION is able to identify more general BCs and handles more specifications. Specifically, LOGION produces about 1 order of magnitude more general BCs than the SOTA approaches. Besides, the results confirm the effectiveness of LISF: LOGION using LISF is faster about 10 times than LOGION using LTL satisfiability checking for large-scale cases.

Our main contributions are summarized as follows.

1. We observe that the SOTA approach tends to identify similar BCs in structure and hypothesize that it is a positive heuristic to identify BCs.
2. To better utilize the structural similarity, we design a novel local search-based approach to identify general BCs on large-scale specifications.
3. We propose an incremental satisfiability filter based on lasso sets to speed up the BC verification.
4. Experimental results confirm our hypotheses and the efficiency and efficacy of our approach.

Compared to the previous conference version (Zhong et al., 2020), this paper extends the contribution as follows:

1. improve local search by strengthening and weakening operators;
2. propose LISF to speed up the BC verification;
3. re-construct the experiments by applying more cases from industry, to better support the efficiency of LOGION.

[1] Our code and datasets are publicly available at https://github.com/sysulic/LOGION-SWLS-TBC-LISF.

The remainder of the paper is organized as follows. We give some background in Section 2. Next, we show how the structural similarity of BCs occurs frequently in Section 3 and describe our approach in detail in Sections 4 and 5. In Section 6, we carry out our experiments to validate our approach and compare it with related approaches. Finally, we discuss related work in Section 7 and make some conclusions in the last section.

2. Background

In this section, we introduce the background of LTL, goal-conflict identification, tree edit distance, and local search. We briefly recall some basic notions for the rest of the paper.

2.1. Linear temporal logic

Linear temporal logic (LTL) (Pnueli, 1977) is suitable for specifying software requirements (Van Lamsweerde et al., 1998). The syntax of LTL for a finite set of propositions $\mathbb{P}$ includes logical connectives ($\wedge$, $\neg$), $\mathbb{B} = \{\bot, \top\}$, and temporal operators (*next* ($\bigcirc$), *until* ($\mathcal{U}$)):

$$\varphi := p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \mathcal{U} \varphi_2,$$

where $p \in \mathbb{P} \cup \mathbb{B}$ and φ_1 and φ_2 are LTL formulae. For brevity, we only consider the above fundamental operators in this paper. Operator *release* ($\mathcal{R}$), *eventually* ($\Diamond$), *always* ($\Box$), and *weak-until* ($\mathcal{W}$) are commonly used, and can be defined by the fundamental operators as $\varphi_1 \mathcal{R} \varphi_2 := \neg(\neg\varphi_1 \mathcal{U} \neg\varphi_2)$, $\Diamond\varphi := \top \mathcal{U} \varphi$, $\Box\varphi := \neg(\top \mathcal{U} \neg\varphi)$, and $\varphi_1 \mathcal{W} \varphi_2 := (\varphi_1 \mathcal{U} \varphi_2) \vee \Box\varphi_1$, respectively. Let φ be an LTL formula, the set of *sub-formulae* $\mathsf{sub}(\varphi)$ of φ is defined recursively as follows: if $\varphi = b$ or $\varphi = p$ with $b \in \mathbb{B}$, $p \in \mathbb{P}$, then $\mathsf{sub}(\varphi) = \{\varphi\}$; if $\varphi = o_1\varphi'$ with $o_1 \in \{\neg, \bigcirc, \Box, \Diamond\}$, then $\mathsf{sub}(\varphi) = \{\varphi\} \cup \mathsf{sub}(\varphi')$; if $\varphi = \varphi'\ o_2\ \varphi''$ with $o_2 \in \{\wedge, \vee, \mathcal{U}, \mathcal{W}, \mathcal{R}\}$, then $\mathsf{sub}(\varphi) = \{\varphi\} \cup \mathsf{sub}(\varphi') \cup \mathsf{sub}(\varphi'')$. $|\varphi|$ denotes the *size* of the formula φ, *i.e.*, the number of temporal operators, logical connectives, and propositions in φ. Every LTL formula corresponds to a *parse tree* (Enderton and Enderton, 2001).

LTL formulae are interpreted over a *linear-time structure*. A linear-time structure is a pair of $W = (S, \varepsilon)$ where S is a infinite-state sequence, and $\varepsilon : S \to 2^{\mathbb{P}}$ is a function mapping each state s_i to a set of propositions. Let W be a linear-time structure, $i \geq 1$ a position, and φ_1, φ_2 LTL formulae. The *satisfaction relation* $\vDash$ is defined as follows:

$W, i \vDash p$	iff	$p \in \varepsilon(s_i)$, where $p \in \mathbb{P}$ or $p = \top$,
$W, i \vDash \neg\phi$	iff	$W, i \nvDash \phi$,
$W, i \vDash \varphi_1 \wedge \varphi_2$	iff	$W, i \vDash \varphi_1$ and $W, i \vDash \varphi_2$,
$W, i \vDash \bigcirc\varphi$	iff	$W, i+1 \vDash \varphi$,
$W, i \vDash \varphi_1\ \mathcal{U}\ \varphi_2$	iff	$\exists k \geq i$ s.t. $W, k \vDash \varphi_2$ and , $\forall i \leq j < k$, $W, j \vDash \varphi_1$.

An LTL formula φ is called *satisfiable* if and only if there is a linear-time structure W, such that $W, 0 \vDash \varphi$. The linear-time structure satisfying φ is called as a *model* of φ. An LTL formula φ *implies* an LTL formula φ', noted $\varphi \to \varphi'$, if the models of φ are also models of φ'. Two LTL formulae φ_1 and φ_2 are *logically equivalent*, denoted by $\varphi_1 \equiv \varphi_2$, if and only if $\varphi_1 \to \varphi_2$ and $\varphi_2 \to \varphi_1$. The LTL satisfiability problem is to check whether an LTL formula is satisfiable, which is PSPACE-complete (Sistla and Clarke, 1985). Recently, LTL satisfiability solvers based on different techniques have been developed. Among these solvers, nuXmv (Cavada et al., 2014) and Aalta (Li et al., 2015) have achieved better performance. A *lasso* is a linear-time structure in the form of uv^{ω}, where u and v are finite-state sequences. v^{ω} is a *loop* and means that v appears in order and infinitely. Checking whether a lasso l is a model of φ is in bilinear time ($O(|l| \cdot |\varphi|)$) (Markey and Schnoebelen, 2003). We call this checking process as *trace checking*.

Let φ be an LTL formula over fundamental operators ($\{\neg, \wedge, \bigcirc, \mathcal{U}\}$) and ψ be a sub-formula of φ. ψ has a *positive polarity* (+) in φ if it appears under an even number of negations, *negative polarity* (−) otherwise. $P_{\varphi}(\psi)$ denotes the polarity of ψ in φ. We use Example 2 to illustrate the polarity.

Example 2 (*Example 1 Cont.*)**.** Let $\varphi = \Box(m \to \bigcirc(\neg p))$. The logically equivalent formula over fundamental operators of φ is $\neg(\top \mathcal{U} \neg(\neg m \vee \bigcirc(\neg p)))$. Let $\psi_1 = m$ and $\psi_2 = \bigcirc(\neg p)$. Then $P_{\varphi}(\psi_1) = -$ and $P_{\varphi}(\psi_2) = +$.

Fisman et al. (2008) introduced *strengthening* and *weakening* of the formula over fundamental operators. The strengthening (*resp.* weakening) formula of φ, denoted by $\varphi^{\oplus}$ (*resp.* $\varphi^{\ominus}$), is replacing ψ with $\bot$ in φ if $P_{\varphi}(\psi) = +$ (*resp.* $P_{\varphi}(\psi) = -$); otherwise replacing ψ with $\top$. We denote a set of $\varphi^{\oplus}$ (*resp.* $\varphi^{\ominus}$) as $\sum_{\varphi}^{\oplus}$ (*resp.* $\sum_{\varphi}^{\ominus}$). $\sum_{\varphi}^{\oplus}$ and $\sum_{\varphi}^{\ominus}$ are computed by applying the above replacing procedure to all sub-formulae with polarity. We use Example 3 to illustrate the strengthening and weakening.

Example 3 (*Example 1 Cont.*)**.** Let $\varphi = \Box(m \to \bigcirc(\neg p))$, then $\sum_{\varphi}^{\oplus} = \{\Box(\bot), \Box(\top \to \bigcirc(\neg p)), \Box(m \to \bot), \Box(m \to \bigcirc(\bot)), \Box(m \to \bigcirc(\neg\top))\}$ and $\sum_{\varphi}^{\ominus} = \{\Box(\top), \Box(\bot \to \bigcirc(\neg p)), \Box(m \to \top), \Box(m \to \bigcirc(\top)), \Box(m \to \bigcirc(\neg\bot))\}$.

Based on the definitions, we have the property as follows.

Property 1. *Let φ be an LTL formula over fundamental operators. $\varphi^{\oplus} \to \varphi$ and $\varphi \to \varphi^{\ominus}$ hold.*

2.2. Goal-conflict identification

In GORE, goals represent mandatory objectives the system needs to fulfill, whereas domain properties are descriptive assertions that characterize the attributes of the problem world. It is unrealistic to require requirement specification to be complete or all goals to be ideally satisfiable, because unanticipated cases may occur. It suggests identifying the conflicts as early as possible, so the goal-conflict identification stage is important in the conflict analysis phase. The *conflict analysis phase* (Van Lamsweerde and Letier, 1998; Van Lamsweerde, 2009) has three main stages:

1. the *identification stage* is to identify condition whose occurrence makes the divergence;
2. the *assessment stage* is to assess and prioritize the identified conflicts according to their likelihood and severity;
3. the *resolution stage* is to resolve the identified conflicts by providing appropriate countermeasures.

In this paper, we focus on identifying divergences, which are captured by *boundary conditions (BCs)* (Definition 1). Intuitively, a BC captures a situation where the goals as a whole are unsatisfiable.

Definition 1. Let $G = \{G_1, \ldots, G_n\}$ be goals and D domain properties. Suppose $D \wedge G$[2] is satisfiable. The goals are said to be divergent in the context of φ if there exists an expression φ, called a *boundary condition (BC)*, such that the following properties hold:

$$D \wedge G \wedge \varphi \to \bot, \qquad \text{(logical inconsistency)}$$

$$D \wedge G_{-i} \wedge \varphi \not\to \bot, \text{ for each } 1 \leq i \leq n, \qquad \text{(minimality)}$$

$$\neg G \not\equiv \varphi, \qquad \text{(non-triviality)}$$

where $G_{-i} = G \setminus \{G_i\}$.

Formal requirements engineering methodologies usually use some formal languages to accurately express domain properties, goals, and BCs, so that efficient (semi-)automatic reasoning can be performed. In this paper, we focus on LTL, which is also widely used in previous works (Degiovanni et al., 2016, 2018b). With the help of an efficient LTL satisfiability solver, we can automatically verify whether a formula is a BC by checking whether it satisfies the properties.

Definition 2. Let S be a set of BCs. A BC $\varphi_i \in S$ is *more general* than another BC $\varphi_j \in S$ if φ_j implies φ_i.

[2] Here we use the set to denote the conjunction of the elements.

Intuitively, a more general BC φ captures all the particular combinations of circumstances captured by the less general BCs than φ. Therefore, it is also important to find more general BCs.

2.3. Tree edit distance

The *tree edit distance* (Zhang and Shasha, 1989) is used to measure the similarity between two ordered labeled trees and has successfully been applied in a wide range of applications. It is the cost of the minimal-cost sequence of node edit operations that transforms one tree into another. For an ordered label tree, there are three node edit operations:

- *relabel* the label of a node in the tree;
- *insert* a node between an existing node and the all children of this node;
- *delete* a non-root node and connect its children to its parent maintaining the order.

There are many different sequences of node edit operations that transform one tree into another. The cost of each node edit operation is 1, and the cost of a sequence is the sum of the cost of its node edit operations. Tree edit distance is the sequence with the minimal cost. Given two trees T_1 and T_2, the tree edit distance is denoted by $\delta(T_1, T_2)$.

In fact, for the trees with large sizes, the tree edit distance becomes unsuitable to represent the difference between these trees. For example, a tree edit distance of 5 means a big gap between two trees with the size 10 but a small gap between two trees with the size 1000. To capture the relative similarity *w.r.t.* the size, the notion of the normalized tree edit distance was proposed in Rico-Juan and Micó (2003), which is defined as Eq. (1).

$$\Delta(T, T') = \frac{\delta(T, T')}{|T| + |T'|} \in [0, 1], \tag{1}$$

where $|T|$ denotes the size of the tree T, *i.e.*, the number of nodes in T.

2.4. Local search

Local search (Hoos and Stützle, 2004) is a kind of meta-heuristic search algorithm. Formally, given a problem instance π, we use $S(\pi)$ to denote its search space, which is the set of all candidate solutions. The *neighborhood function* $N : S(\pi) \to 2^{S(\pi)}$ maps each candidate solution to its neighbors. The elements of $N(s)$ are called the *neighbors* of s. The *objective function* $f : S(\pi) \to \mathbb{R}$ is a mapping of candidate solutions to their objective function value. Typically, a local search algorithm constructs an initial candidate solution and modifies it iteratively. At each search step, the algorithm evaluates the neighbors of the current candidate solution by the objective function and move to the best neighbor. The search procedure terminates when it runs out of time or the best solution found has not been improved in a given number of steps.

Due to the limited amount of local information when choosing a neighbor to move to, the local search suffers from the cycle problem, that is, some candidate solutions of high quality are being frequently revisited. *Tabu search* (Glover, 1986; Hansen and Jaumard, 1990) is a fundamentally different approach to reduce cycling which forbids steps to recently visited candidate solutions. The most basic and commonly used version of tabu search involves an iterative improvement algorithm, supplemented by a short-term memory component M. This component keeps track of the most recent T solutions visited, where T is called the *tabu tenure*. In each search step, the algorithm chooses the best neighbor in $N(s) \setminus M$, where s is the current candidate solution.

3. Structural similarity of boundary conditions

In this section, we show the structural similarity of BCs in cases.

Table 1
The details of each case include the numbers of domain properties (#Dom), goals (#Goal), propositions (#Prop), and the total size of all formulae (Size) for the specification of each case.

Case	#Dom	#Goal	#Prop	Size
RetractionPattern1 (RP1)	0	2	2	9
RetractionPattern2 (RP2)	0	2	4	10
Elevator (Ele)	1	1	3	10
TCP	0	2	3	14
AchieveAvoidPattern (AAP)	1	2	4	15
MinePump (MP)	1	2	3	21
ATM	1	2	3	22
Rail Road Crossing System (RRCS)	2	2	5	22
Telephone (Tel)	3	2	4	31
London Ambulance Service (LAS)	0	5	7	32
Prioritized Arbiter (PA)	1	6	6	57
Round Robin Arbiter (RRA)	6	3	4	77
Simple Arbiter (SA)	4	3	6	84
Load Balancer (LB)	3	7	5	85
LiftController (LC)	7	8	6	124
ARM's Advanced Microcontroller Bus Architecture (AMBA)	6	21	16	415

3.1. Structural similarity

In order to capture the structural similarity of LTL formulae, we first borrow the notion of the similarity on *ordered label trees*. It is notable that every LTL formula corresponds to a parse tree which is also an ordered label tree (Enderton and Enderton, 2001). We use T_ϕ to denote the parse tree of the formula ϕ. Note that the size of the formula ϕ equals to the size of its parse tree T_ϕ, *i.e.*, $|T_\phi| = |\phi|$. The parse tree of the formula $\Box(h \to \bigcirc(p))$ is shown in Fig. 1(a).

Next, we define the distance between two formulae as the tree edit distance between their corresponding parse trees. Formally, for two formulae ϕ and ϕ', we use $\delta(\phi, \phi') = \delta(T_\phi, T_{\phi'})$ to denote the formula distance between ϕ and ϕ'. As BCs in different specifications may differ seriously on the formula size, we also consider the relative distance between two formulae *w.r.t.* their sizes. Similarly, we use $\Delta(\phi, \phi') = \Delta(T_\phi, T_{\phi'})$ to denote the normalized formula distance between ϕ and ϕ'.

Intuitively, the distance between two formulae indicates their divergence on the structure: the distance is bigger, the divergence is bigger. Indeed, the formula distance also has the property of the tree edit distance:

- $\delta(\phi, \phi) = \Delta(\phi, \phi) = 0$,
- $\Delta(\phi, \phi') \in [0, 1]$.

We use Example 4 to illustrate the formula distance and the normalized formula distance.

Example 4 (*Example 1 Cont.*)**.** Let $\varphi_1 = \Box(h \to \bigcirc(p))$, $\varphi_2 = \Box(h \wedge m)$ and $\varphi_3 = \Diamond(h \wedge m)$. For formula distance, $\delta(\varphi_1, \varphi_2) = 3$ and $\delta(\varphi_2, \varphi_3) = 1$. For normalized formula distance, $\Delta(\varphi_1, \varphi_2) = 0.333$ and $\Delta(\varphi_2, \varphi_3) = 0.125$.

3.2. Experiment about structural similarity

Here we have our first research question:

RQ 1. *How frequently similar BC pairs do occur?*

We use 16 artificial cases introduced by Degiovanni et al. (2018b). Table 1 summarizes the details for the specification of each case. We generated BCs by applying the existing approach (Degiovanni et al., 2018b) for 24 h (10 runs in parallel). We are not using the approach (Degiovanni et al., 2016). The reason is that it is deterministic and the number of identified BCs is small, which easily introduces errors in statistical results. In particular, it only identifies 1 BC in some cases, where it is impossible to compute the similarity.

In order to explore how frequently similar BC pairs occur, for each case, we compute the proportion of BCs which have at least a similar BC whose formula distance is not greater than l ($l = 1, 2, 3$), on the total number of BCs. Formally, we use $\%BC(\delta \leq l)$ to denote the proportion defined as Eq. (2).

$$\%BC(\delta \leq l) = \frac{\#\{BC | \exists BC'.\delta(BC, BC') \leq l\}}{\#\text{total BC}}. \quad (2)$$

We also compute the average number of similar BCs of each BC whose distance is not greater than l. Formally, given a set of BCs and a BC ϕ, we use $sim(\phi, l) = \{\phi' | \delta(\phi, \phi') \leq l\}$ to denote the set of BCs whose formula distance to ϕ is not greater than l. We use $\#sim(\delta \leq l)$ to denote the average number of similar BCs defined as Eq. (3).

$$\#sim(\delta \leq l) = \frac{\sum \#sim(\phi, l)}{\#\text{total BC}}. \quad (3)$$

The results are shown in Table 2. Taking the example of LAS, for each BC ϕ, there are 18.4 BCs on average which differ from ϕ only on one or two propositions or operators. We can observe that in most cases except for LC and AMBA, there are more than 80% BCs which can be obtained by applying two or three formula edit operations from another BC. In these cases, each BC has on average more than one BC that differs at most only in propositions or operators.

Besides the formula distance, we also consider the normalized formula distance in the cases of different upper bound k ($k = 0.1, 0.2, 0.3$). The results are also shown in Table 2. For example, in RRCS, every BC ϕ on average has 85.8 BCs which has a normalized formula distance to ϕ less than 0.2. The results illustrate that in each case, there are more than 99% BCs having a relatively similar BC whose normalized formula distance is at most 0.3. The results also show that each BC on average has more than 10 similar BCs whose normalized formula distance is at most 0.2.

Next, in order to explore how close the similar BC pairs are, for each case, we compute the average minimum distance between BCs and their most similar BCs ("avg min" in Table 2). It shows that in most cases, most BCs have a similar BC whose formula distance is less than 2.

To answer RQ 1, in most cases, there are more than 90% BCs which have both an absolutely similar BC ($\delta \leq 3$) a relatively similar BC ($\Delta \leq 0.3$). Similar BC pairs are common in cases solved by the approaches (Degiovanni et al., 2018b). Since the approaches (Degiovanni et al., 2018b) greatly increase the efficiency of identifying BCs compared to previous approaches, we have reason to hypothesize that its performance improvement comes from its identification of similar BCs. It inspires us to design an local search algorithm to identify BC more efficiently. Because the search mechanism of the local search algorithm is to start from an initial solution and gradually search for solutions near the initial solution through small modifications, we can design a modifying strategy so that the algorithm can find the structurally similar BCs after finding a BC.

3.3. Discussion

Although the observation is derived from the results of the approaches (Degiovanni et al., 2018b), our observation is somewhat generalizable, because the approaches (Degiovanni et al., 2016, 2018b) do not use the observation to design their algorithms. Although there are some genetic operators showing that the GA-based approach (Degiovanni et al., 2018b) can search syntactically similar formulae, the GA-based approach does not take full advantage of this observation due to the diversity and randomness of the evolution process.

4. Identifying general boundary conditions via local search

We develop a local search algorithm, namely LOGION, to find general BCs where the mechanism of local search can effectively utilize the structural similarity of BCs. We first introduce the components, then give the entire description.

4.1. Initialization

To find BCs quickly, the initial formula in the local search algorithm should be as "close" to a BC as possible. So we propose $\neg(G_1 \wedge \cdots \wedge G_n)$, called *trivial condition*, as an initial formula. It is easy to construct and "closest" to a BC because it only violates non-triviality (Definition 1). Besides, it is the most general BCs except that it does not satisfy the non-triviality. Therefore, another motivation is that we potentially identify a more general BC in a short time starting from the formula that is close to the most general BC. We use an example to illustrate the initial formula.

Example 5 (*Example 1 Cont.*)**.** Consider Example 1, the initial formula is $\neg(\square(h \rightarrow \bigcirc(p)) \wedge \square(m \rightarrow \bigcirc(\neg p)))$.

4.2. Neighborhood function

In what follows, we define the neighborhood function. The search space is composed of LTL formulae. To define the neighbors of an LTL formula, we propose *formula edit operations* (Definition 3) that mutate an LTL formula to some others in syntax.

Definition 3. Let o_1, o_1' be two different LTL unary operators (*i.e.*, $o_1, o_1' \in \{\neg, \bigcirc, \square, \Diamond\}$), o_2, o_2' two different LTL binary operators (*i.e.*, $o_2, o_2' \in \{\wedge, \vee, \mathcal{U}, \mathcal{W}, \mathcal{R}\}$), $\mathbb{P}$ a set of propositions, and $\mathbb{B} = \{\top, \bot\}$. The three *formula edit operations* for an LTL formula ψ is defined as follows. We use ψ' to denote the new formula.

1. rename
   (a) $\psi' = p'$, when $\psi = p$, $p \in \mathbb{P} \cup \mathbb{B}$, $p' \in \mathbb{P} \cup \mathbb{B}$;
   (b) $\psi' = o_1' \psi_1$, when $\psi = o_1 \psi_1$;
   (c) $\psi' = \psi_1\ o_2'\ \psi_2$, when $\psi = \psi_1\ o_2\ \psi_2$.
2. insert
   (a) $\psi' = o_1' \psi$;
   (b) $\psi' = \psi\ o_2'\ p$ or $p\ o_2'\ \psi$, when $p \in \mathbb{P} \cup \mathbb{B}$.
3. delete
   (a) $\psi' = \psi_1$, when $\psi = o_1 \psi_1$;
   (b) $\psi' = \psi_1$ or $\psi' = \psi_2$, when $\psi = \psi_1\ o_2\ \psi_2$.

Intuitively, rename replaces the top symbol (literal or operator) of T_ψ with other same type symbol; insert adds a new LTL operator to be top operator of T_ψ; delete removes the top operator of T_ψ. Based on the formula edit operations, we define our neighborhood function as follows.

Definition 4. Given an LTL formula φ as input, the *neighborhood function w.r.t.* φ, denoted by $\mathrm{N}(\varphi)$, returns all formulae obtained by applying a formula editing operation to a sub-formulae of φ.

Since there are too many neighbors for φ, to efficiently select a good candidate BC, we bound the number of neighbors to K in $\mathrm{N}(\varphi)$, denoted by $\mathrm{N}_K(\varphi)$, where K is a hyperparameter. Specifically, we first select K sub-formulae of φ uniformly at random, then, for each sub-formulae, select a formula edit operation uniformly at random to modify it. We use an example to illustrate $\mathrm{N}_K(\varphi)$.

Example 6 (*Example 1 Cont.*)**.** Considering a formula $\varphi = \square(h \rightarrow \bigcirc(p))$ and $K = 3$, Fig. 1(b), (c), and (d) show the result of rename, insert, and delete, respectively.

In order to alleviate visiting some formulae frequently, *i.e.*, the cycling problem, we employ the tabu search strategy. An LTL formula is called a T-tabu formula if and only if it has been visited during the last T search steps, where T is tabu tenure and is a hyperparameter. The neighborhood of φ *w.r.t.* T-tabu, denoted by $\mathrm{N}_K(\varphi, T)$, is defined as $\{\beta \mid \beta \in \mathrm{N}_K(\varphi)$ and β is not a T-tabu formula$\}$

Table 2
Experimental results on the structural similarity of BCs computed by the approach (Degiovanni et al., 2018b). "$\%BC(\delta \le l)$" and "$\%BC(\Delta \le k)$" mean the proportion of BCs which have at least a similar BC whose formula distance is at most l and whose normalized formula distance is at most k, respectively. "#sim($\delta \le l$)" and "#sim($\Delta \le l$)" mean the average number of similar BCs of each BC whose formula distance are at most l and whose normalized formula distance is at most k. "avg min" means the average distance between BCs and their most similar ones. "#total BC" stands for the total number of different BCs identified by the approach (Degiovanni et al., 2018b) for 24 h (10 runs in parallel).

Case	$\%BC(\delta \le l)$			#sim($\delta \le l$)			$\%BC(\Delta \le k)$			#sim($\Delta \le k$)			avg min	#total BC
	l = 1	l = 2	l = 3	l = 1	l = 2	l = 3	k = 0.1	k = 0.2	k = 0.3	k = 0.1	k = 0.2	k = 0.3		
RP1	87.8%	95.8%	97.0%	3.6	12.8	26.2	97.7%	99.5%	100.0%	28.7	96.2	306.4	1.3	790
RP2	94.8%	97.9%	98.5%	4.1	12.5	22.6	97.9%	99.8%	100.0%	13.5	40.5	125.9	1.1	477
Ele	77.2%	82.5%	82.5%	2.1	4.7	5.9	82.5%	93.0%	100.0%	5.3	11.9	21.6	2.4	57
TCP	88.0%	92.8%	94.2%	4.0	13.2	22.5	95.5%	99.4%	100.0%	21.5	59.6	134.2	1.5	359
AAP	76.1%	87.5%	94.0%	2.6	9.2	20.6	76.1%	95.1%	100.0%	2.3	10.6	32.9	1.5	184
MP	75.1%	83.7%	87.8%	2.0	6.2	13.3	83.9%	97.0%	100.0%	3.7	15.0	47.0	1.7	361
ATM	80.5%	89.8%	93.1%	2.6	6.8	12.4	92.2%	98.8%	100.0%	6.6	21.5	87.0	1.5	334
RRCS	81.9%	90.6%	93.3%	2.5	7.6	13.9	96.2%	100.0%	100.0%	16.0	85.8	218.9	1.5	552
Tel	88.8%	95.9%	98.2%	2.9	8.1	15.5	94.7%	98.8%	100.0%	5.3	23.2	63.2	1.2	170
LAS	97.1%	98.5%	98.5%	5.7	18.4	29.4	98.5%	100.0%	100.0%	30.7	46.8	96.2	1.1	136
PA	87.2%	94.4%	95.7%	3.5	15.7	39.7	97.5%	99.9%	100.0%	64.0	175.4	527.2	1.4	1495
RRA	85.5%	94.3%	96.4%	2.8	8.6	16.2	98.8%	99.9%	100.0%	28.5	110.3	359.0	1.3	855
SA	54.9%	81.7%	83.1%	2.1	7.4	11.5	90.1%	100.0%	100.0%	14.1	19.4	31.2	2.7	71
LB	64.0%	73.0%	80.9%	1.3	4.4	7.5	87.6%	97.8%	100.0%	12.9	31.9	53.7	3.0	89
LC	37.6%	64.3%	70.5%	0.6	1.7	3.2	77.6%	94.8%	99.0%	4.1	12.9	32.3	3.1	210
AMBA	24.4%	45.5%	50.7%	0.3	0.7	0.9	94.8%	98.1%	100.0%	10.2	33.1	53.4	5.8	213

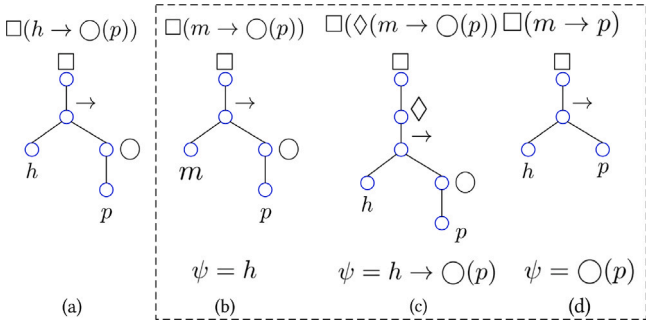

Fig. 1. An example of getting neighbors of $\Box(h \to \bigcirc(p))$. (a) $\Box(h \to \bigcirc(p))$ and its parse tree. (b) The neighbor formula after renaming h to m and its parse tree. (c) The neighbor formula after inserting $\Diamond$ and its parse tree. (d) The neighbor formula after deleting $\bigcirc$ and its parse tree.

4.3. Strengthening and weakening operators

The neighborhood function $N_k(\varphi)$ captures the LTL formulae with similar syntax structures of φ as neighbors. We also need to capture the semantics of the formula to make the LOGION have a search bias towards more general BCs. The idea is, for an LTL formula φ, we use the strengthening and weakening operations to find its $\varphi^{\oplus}$ and $\varphi^{\ominus}$. If φ is BC, then we search $\varphi^{\ominus}$ which is also a BC. The $\varphi^{\ominus}$ is more general BC than φ since $\varphi \to \varphi^{\ominus}$. If φ violates logical inconsistency, we look for $\varphi^{\oplus}$ which has $D \wedge G \wedge \varphi^{\oplus} \to \bot$, *i.e.*, $\varphi^{\oplus}$ holds logical inconsistency. If φ violates minimality, we find $\varphi^{\ominus}$ which holds $\forall G_i, D \wedge G_{-i} \wedge \varphi^{\ominus} \not\to \bot$, *i.e.*, $\varphi^{\ominus}$ holds minimality. Based on this idea, we design *Strengthening and Weakening Operators of Neighbor (SWON)* to operate neighbors.

The pseudo-code of SWON is shown in Algorithm 1 which takes $N_K(\varphi, T)$ as input Δ. For each formula $\eta \in N_K(\varphi, T)$, we use the lasso-based incremental satisfiability filter in Section 5 to verify whether η is a BC from Definition 1 (line 3). If η is a BC, then SWON tries to find a BC from the weakening formulae of η, and append it to Δ (line 4–7). If η violates logical inconsistency, then SWON tries to find the first formula that satisfies logical inconsistency in $\sum_{\eta}^{\oplus}$ and append it to Δ (line 10–11). Similarly, if η violates minimality, SWON tries to find the first formula that satisfies minimality in $\sum_{\eta}^{\ominus}$ and append it to Δ (line 14–15). Finally, SWON returns the updated neighbor set Δ (line 16). We use an example to illustrate Algorithm 1.

Algorithm 1: SWON

Input: $N_K(\varphi, T)$, domain properties D, and goals G.
Output: the updated neighbor set Δ.

```
1  Δ ← N_K(φ, T)
2  for each neighbor η ∈ N_K(φ, T) do
3      check whether η is a BC from Definition 1 by the filter in
         Section 5;
4      if η is a BC then
5          Σ_η^⊖ ← the weakening set of η;
6          if ∃η^⊖ ∈ Σ_η^⊖, η^⊖ is a BC then
7              add η^⊖ to Δ;
8      else if η violates logical inconsistency then
9          Σ_η^⊕ ← the strengthening set of η;
10         if ∃η^⊕ ∈ Σ_η^⊕, η^⊕ satisfies logical inconsistency then
11             add η^⊕ to Δ;
12     else if η violates minimality then
13         Σ_η^⊖ ← the weakening set of η;
14         if ∃η^⊖ ∈ Σ_η^⊖, η^⊖ satisfies minimality then
15             add η^⊖ to Δ;
16 return Δ;
```

Example 7 (*Example 1 Cont.*). Let $N_3(\varphi, T) = \{\eta_1 = \bigcirc(h \wedge m \wedge p), \eta_2 = m \vee (h \wedge m \wedge p), \eta_3 = m \wedge \bigcirc(h \wedge m \wedge p)\}$. η_1 is a BC and the $\eta_1^{\ominus} = \bigcirc(h \wedge m \wedge \top)$ is also a BC, then SWON adds $\eta_1^{\ominus}$ to Δ. η_2 violates logical inconsistency. The $\eta_2^{\oplus} = \bot \vee (h \wedge m \wedge p)$ which holds logical inconsistency. So SWON adds $\eta_2^{\oplus}$ to Δ. η_3 violates minimality. The $\eta_3^{\ominus} = m \wedge \bigcirc(h \wedge m \wedge \top)$ which holds minimality, then SWON adds $\eta_3^{\ominus}$ to Δ. Finally, SWON returns $\Delta = \{\eta_1, \eta_2, \eta_3, \eta_1^{\ominus}, \eta_2^{\oplus}, \eta_3^{\ominus}\}$.

4.4. Neighbor selection strategy

In each iteration, the neighbor set Δ has multiple formulae. We propose a *Neighbor Selection Strategy (NSS)* to select the best formula in Δ for the next iteration. The NSS considers two cases separately. If there are multiple BCs in Δ, then NSS chooses the shortest BC because it may be easy for humans to understand. There is no BC in Δ, otherwise, NSS randomly selects a formula from the set of formulae Π "closest" to BC. The Π is constructed in the Δ by the following steps.

Algorithm 2: LOGION

Input: the domain properties D and goals $G_1, \ldots, G_n$.
Output: a set of identified BCs $\mathcal{B}$.

```
1  B ← ∅;
2  φ* ← ¬(G_1 ∧ ··· ∧ G_n);
3  while the cutoff time is not reached do
4      if N_K(φ*, T) is not an empty set then
5          Δ ← SWON(N_K(φ*, T), D, {G_1, ..., G_n});
6          B, φ* ← NSS(B, Δ);
7      else
8          φ* ← the formula earliest added to T-tabu;
9      T-tabu update;
10 return B;
```

1. NSS selects the formulae holding the maximum number G_{-i} subject to $D \wedge G_{-i} \wedge \varphi \not\vdash \bot$ in Δ as Π_1.
2. If there is a formula holding logical inconsistency property in Π_1, NSS deletes formulae without this property in Π_1 as Π_2. Otherwise, Π_1 becomes Π_2.
3. If there is a formula holding non-triviality property in Π_2, NSS deletes formulae without this property in Π_2 as Π. Otherwise, Π_2 becomes Π.

4.5. Description of LOGION

Now, we give the entire description of LOGION (Algorithm 2) based on the components described in this section. Overall, LOGION uses the trivial condition as the current formula at the beginning (line 2), and searches for a local optimal candidate BC, denoted by φ^*, in each iteration (line 3–9). Note that although φ^* may not be a BC, φ^* is closest to a BC because of the NSS. Finally, LOGION returns a set of identified BCs $\mathcal{B}$ when the algorithm reaches a time limit (line 10).

At each iteration, If $\mathrm{N}_K(\varphi^*, T)$ is an empty set, it means all neighbors of φ^* have been visited in T steps. Then LOGION resets φ^* as the formula earliest added to T-tabu (line 8). The motivation is to search for a new branch starting from a formula that is more syntactically different from the current candidate BC. Otherwise, LOGION updates the Δ based on the strengthening and weakening operators (line 5). Then LOGION walks to the neighbor that is more likely to be a more general BC. After that, LOGION updates the T-tabu, where the formula earliest added is released and φ^* is added (line 9).

5. Lasso-based incremental satisfiability filter

It is a bottleneck to verify whether an LTL formula is a BC for the existing approaches. The reason is that the existing approaches simply invoke an LTL satisfiability solver multiple times to verify a BC, yet the LTL satisfiability problem is PSPACE-complete (Sistla and Clarke, 1985). Specifically, it requires $|G| + 2$ calls of a satisfiability solver to check if a formula φ is a BC: one for logical inconsistency, $|G|$ calls for minimality, and one for non-triviality (check whether $\neg((\neg G \rightarrow \varphi) \wedge (\varphi \rightarrow \neg G))$ is satisfiable). Consequently, the performance of BC verification will be remarkably boosted if we can efficiently check the satisfiability of formulae.

We observe that, given different LTL formulae to be verified, the checked formulae are in a form of $\psi \wedge \varphi$ in which ψ is fixed ($Dom \wedge G$ for logical inconsistency, $Dom \wedge G_{-i}$ for minimality, and G for non-triviality), and φ is a formula to be verified (for the case $\neg G \wedge \neg\varphi$, it is similar). It is obvious that if φ is satisfied by a model of ψ, then $\psi \wedge \varphi$ is satisfiable. In other words, in the best case, we can check whether a lasso is a model (trace checking) instead of checking LTL satisfiability. Note that it is in bilinear time to check whether a lasso is a model of an LTL formula (Markey and Schnoebelen, 2003), which significantly reduces the time cost compared against the complete LTL satisfiability checking.

Algorithm 3: LISF

Input: ψ, φ and lasso set $\sum_\psi$
Output: whether $\psi \wedge \varphi$ is SAT and an updated $\sum_\psi$

```
1  foreach l ∈ Σ_ψ do
2      if TraceCheck(l, φ) == True then
3          /* l is a model of ψ ∧ φ */
4          return SAT, Σ_ψ;
5  (imp, l) ← ModelCheck(ψ, ¬φ);
6  if imp == True then
7      /* ψ → ¬φ, i.e., ψ ∧ φ is UNSAT */
8      return UNSAT, Σ_ψ;
9  else
10     /* l is a counterexample trace, i.e., l ⊨ ψ ∧ φ */
11     Σ_ψ ← Σ_ψ ∪ {l};
12     return SAT, Σ_ψ;
```

Based on this idea, We propose a *Lasso-based Incremental Satisfiability Filter (LISF)*, as shown in Algorithm 3. In this way, we maintain the global set $\sum_\psi$ that stores some lassos satisfying ψ. $\sum_\psi$ is initialized as an empty set. LISF has two procedures, the light-weight pre-checking (line 1–4) and the complete satisfiability checking (line 5–12). As the light-weight pre-checking, LISF tries to find a lasso l subject to $l \vDash \varphi$ by trace checking (TraceCheck), which means that $\psi \wedge \varphi$ is satisfiable (line 1–4). Otherwise, LISF will enter a complete satisfiability checking. Specifically, it employs an LTL satisfiability solver to check $\psi \rightarrow \neg\varphi$. If $\psi \rightarrow \neg\varphi$, then $\psi \wedge \varphi$ is unsatisfiable. Otherwise, the solver returns a counterexample trace l which is a new lasso of ψ. LISF adds this lasso into $\sum_\psi$.

We improve the BC checking with employing LISF rather than directly an LTL satisfiability solver. At the beginning of the local search, the lasso sets are small, in consequence, it has to invoke an LTL satisfiability solver frequently. But with the lasso sets growing, it becomes simpler and simpler to find a lasso to hit the candidate BCs. It is worth noting that the large size of the lasso set also reduces the performance. Thus, we employ the last recently used strategy to maintain the lasso set and set the size of $\sum_\psi$ to η which is a hyperparameter. The following example shows the verification process of LISF.

Example 8 (*Example 1 Cont.*). Consider two formulae $\varphi_1 = \neg(\square(m \rightarrow \bigcirc p) \wedge \square(m \rightarrow \bigcirc\neg p)), \varphi_2 = \neg(\square\lozenge(h \rightarrow \bigcirc p) \wedge \square(m \rightarrow \bigcirc\neg p))$. Let $\psi = Dom \wedge G_{-2}$ and $\sum_\psi = \emptyset$. LISF$(\psi, \varphi_1, \emptyset)$ returns $\psi \wedge \varphi_1$ is satisfiable and the lasso $l = s_0(s_1 s_2 s_3)^\omega$ of $D \wedge G_{-2}$, where $s_0 = \{\neg p, \neg h, \neg m\}$, $s_1 = \{\neg p, \neg h, m\}$, $s_2 = \{p, \neg h, m\}$, $s_3 = \{\neg p, \neg h, m\}$. Then LISF$(\psi, \varphi_2, \{l\})$ directly returns that $\psi \wedge \varphi_2$ is satisfiable because $l \vDash \psi \wedge \varphi_2$ by trace checking.

6. Experiments

In this section, we conduct extensive experiments on a broad range of cases to evaluate the effectiveness of LOGION. We start by presenting two research questions and presenting the experimental preliminaries. Then, we show the experimental results and give some discussions about the empirical results.

As a start, we propose the following research questions to evaluate LOGION.

RQ 2. *How does LOGION compare against the Competitors?*

RQ 3. *What are the advantages of our local search algorithm and LISF?*

Algorithm 4: LOGION-simp

Input: the domain properties D and goals $G_1, \ldots, G_n$.
Output: a set of identified BCs $\mathcal{B}$.

```
B ← ∅;
φ* ← random proposition;
while the cutoff time is not reached do
    if N_K(φ*, T) is not an empty set then
        B ← B ∪ all BCs in N_K(φ*, T);
        φ* ← the formula in N_K(φ*, T) with highest score by f;
    else
        φ* ← the formula earliest added to T-tabu;
    T-tabu update;
return B;
```

6.1. State-of-the-art competitors

We compare LOGION against two SOTA approaches. Both of them employ Aalta (Li et al., 2019) as the LTL satisfiability solver.

GA[3] (Degiovanni et al., 2018b). GA is based on a genetic algorithm. It is effective in finding multiple BCs in different forms since it applies suitable crossover and mutation operators.

Tab[4] (Degiovanni et al., 2016). Tab is a deterministic algorithm based on tableaux. It constructs paths that "escape" from the tableau structure and generates a small number of BCs.

We also implement a *simple local search* algorithm as baseline to evaluate our local search algorithm, denoted by LOGION-simp. It uses random propositions as initialization formulae. Its neighbor function is consistent with LOGION, and the objective function is consistent with GA. The pseudo-code of LOGION-simp is shown in Algorithm 4.

6.2. Benchmarks

Degiovanni et al. (2018b) introduced the 16 artificial cases shown in Table 1. Moreover, to evaluate the practicality of the approaches, we introduce two industrial cases, denoted by AMBA and Genbuf. These industrial cases are larger scale (more domain properties and goals) than the artificial cases and therefore more difficult to solve.

AMBA (Ltd, 1999). AMBA is the ARM's Advanced Microcontroller Bus Architecture bus protocol. It gives a formal specification of the arbiter. AMBA uses 2 masters, denoted by AMBA02.

Note that the case named as "AMBA" in Table 1 is the simplified version of AMBA02.

Genbuf (Bloem et al., 2007). Genbuf is a generalized buffer that communicates with n senders and two receivers using two different four-stage handshake protocols. Genbuf uses 1 to 5 senders, denoted by Genbuf01 to Genbuf05, respectively.

For the industrial cases, all the existing approaches (including LOGION) only solve AMBA02 and Genbuf01 to Genbuf05. The larger scale AMBA and Genbuf all methods fail to find BCs. Therefore, the experimental table in this paper only shows the results of AMBA02 and Genbuf01 to Genbuf05. The details of AMBA02 and Genbuf01 to Genbuf05 are shown in Table 3. The different sizes of these cases result from the number of related components (masters for AMBA and senders for Genbuf). As the number of components increases, new goals and domain properties need to be added to describe the constraints of these additional components. Therefore, the size of the goals and domain properties of these components also increases.

Table 3
The details of the industrial cases.

Cases	#Dom	#Goal	#Prop	Size
AMBA02	11	62	26	713
Genbuf01	16	38	14	397
Genbuf02	19	44	16	457
Genbuf03	22	54	19	554
Genbuf04	25	62	21	625
Genbuf05	28	74	24	746

6.3. Experimental settings

We implement LOGION and its variants with Java Meta-heuristics Search Framework (De Beukelaer et al., 2017), integrating the LTL2Büchi library (Giannakopoulou and Lerda, 2002) to parse LTL requirement specifications. The implementation of LISF is based on nuXmv (Cavada et al., 2014) because Aalta cannot provide a satisfactory trace if the LTL formula is satisfiable. To ensure the fairness of the LTL satisfiability solver, we implemented Tab and GA equipped with nuXmv (Tab-nu and GA-nu) to demonstrate the effects of different satisfiability solvers.

We set the hyperparameter as follows. For LOGION and its variants, the tabu tenure T is set to 4, the upper bound of the number of neighbors K is set to 50, and the size η of $\sum_{D \wedge G}$ is set to 10. For GA, we use the hyperparameters provided in their paper. For the cases that were not discussed in their paper (industrial cases), we use the automated hyper-parameter optimizer Smart-Tuner[5] to select 3 sensitive hyperparameters n_c, n_p, and p_m (Degiovanni et al., 2018b). The n_c is the maximum number of genes per chromosome, the n_p is the size of the population maintained per iteration, and the p_m is the probability of mutation operator. The optimal (n_c, n_p, p_m) is $(265, 291, 15\%)$ in AMBA02, $(88, 257, 15\%)$ in Genbuf01 and Genbuf02, $(284, 50, 15\%)$ in Genbuf03 to Genbuf05.

We evaluate approaches using the following metrics. The first is the number of averaged BCs identified, denote by #bc. A larger #bc. shows a stronger ability to identify BCs. The second is #gen., which shows the analysis of the overlapping of the BCs produced by *multiple* approaches. The calculation of #gen. is as follows. Suppose there are n approaches to identifying BCs to be evaluated, and the set of identified BCs is denoted by $B_1, B_2, \ldots, B_n$. #gen. is the size of the set of more general BCs of corresponding approaches. The #gen. of the approach i is computed as (4).

$$\#\text{gen.} = |\{\phi \in B_i | \nexists \psi \in \bigcup_{j=1}^{n} B_j, \phi \rightarrow \psi\}| \tag{4}$$

Intuitively, BCs in the set of more general BCs of corresponding approaches ($\{\phi \in B_i | \nexists \psi \in \bigcup_{j=1}^{n} B_j, \phi \rightarrow \psi\}$) do not overlap each other. Therefore, a larger #gen. shows a stronger ability to identify more general BCs. The third is the number of successful runs (out of 10 runs) to show the robustness of the approaches, denote by #suc.

All the experiments are run on an Intel E7 2.13 GHz with 128 GB memory under running Ubuntu 16.04. The cutoff time was set to 1 h. Each case ran 10 times and we reported mean data. Table 4 lists all the approaches that appear in this experiment, along with their configurations.

6.4. Comparison with state-of-the-art approaches

Since our approach uses nuXmv as the LTL satisfiability solver, for fairness, we first explore the impact of different LTL satisfiability solvers on Tab and GA. The results are shown in Table 5. Tab and Tab-nu identify exactly the same BCs, only Tab is slightly better than

[3] http://dc.exa.unrc.edu.ar/staff/rdegiovanni/ASE2018.html.
[4] http://dc.exa.unrc.edu.ar/staff/rdegiovanni/ase2016.html.
[5] http://smart-tuner.com/.

Table 4
The details of the configuration of approach.

Approaches	Technology	BC checker	Termination condition
Tab (Degiovanni et al., 2016)	tableaux	Aalta	1 h
Tab-nu	tableaux	nuXmv	1 h
GA (Degiovanni et al., 2018b)	genetic algorithm	Aalta	1 h
GA-nu	genetic algorithm	nuXmv	1 h
GA-50Ite	genetic algorithm	Aalta	50 iterations
LOGION	local search	LISF	1 h
LOGION-Aa	local search	Aalta	1 h
LOGION-nu	local search	nuXmv	1 h
LOGION-simp	simp local search	LISF	1 h
LOGION-simp-Aa	simp local search	Aalta	1 h

Table 5
Experimental results compared with different LTL satisfiability solvers. **Bold** numbers mark better results. The computation of #gen. of Tab and Tab-nu only takes into account BCs identified by Tab and Tab-nu. The computation of #gen. of GA and GA-nu is similar. If an approach fails in all 10 runs, "#bc" and "#gen." are marked by "N/A". We omitted the benchmarks where BCs ware not identified by all approaches.

Cases	Tab			Tab-nu			GA			GA-nu		
	#bc	#gen.	time	#bc	#gen.	time	#bc	#gen.	#suc.	#bc	#gen.	#suc.
RP1	1.0	1.0	**0.1**	1.0	1.0	0.9	**77.7**	**8.4**	10	73.9	6.7	10
RP2	1.0	1.0	**0.7**	1.0	1.0	0.9	**90.6**	**4.3**	10	58.6	3.3	10
Ele	1.0	1.0	**0.9**	1.0	1.0	1.3	13.7	**5.5**	10	**14.4**	3.3	10
TCP	2.0	2.0	**1.4**	2.0	2.0	2.0	37.3	**10.8**	10	**41.0**	6.2	10
AAP	4.0	4.0	**0.5**	4.0	4.0	2.0	33.2	**5.0**	10	**33.6**	3.9	10
MP	1.0	1.0	**0.0**	1.0	1.0	0.2	**28.3**	4.8	10	26.6	**6.0**	10
ATM	3.0	2.0	**2.5**	3.0	2.0	3.9	78.9	10.1	10	**85.0**	**14.1**	10
RRCS	1.0	1.0	**0.7**	1.0	1.0	1.8	**74.9**	**13.6**	10	34.3	6.0	10
Tel	1.0	1.0	**8.8**	1.0	1.0	10.0	25.6	4.5	10	**25.8**	**5.0**	10
LAS	1.0	1.0	**3.6**	1.0	1.0	4.5	**39.0**	**4.0**	**2**	N/A	N/A	0
PA	N/A	N/A	>1 day	N/A	N/A	>1 day	**254.0**	**5.6**	**1**	N/A	N/A	0

Tab-nu in terms of time cost. GA can solve two more cases than GA-nu, and it also has advantages in BC number and BC quality. Overall, using nuXmv reduces the performance of Tab and GA-nu. To show the SOTA results, we compare our approach with their advantageous version, *i.e.*, equipped with Aalta as the LTL satisfiability solver.

Table 6 shows the comparison results. On the number of BCs identified ("#bc"), LOGION outperforms Tab and GA by one order of magnitude in most cases. Tab only solves the first 10 small cases, and each case can find up to 4 BCs. It is due to the fact that Tab is so slow that fewer formulae are constructed for BC verification. GA is also a search-based algorithm like LOGION, but its performance is much poorer than LOGION. It indicates that the local search algorithm exhibits a faster search process because it captures the structural similarity of BCs: once it finds a BC, it can find others in its neighbors.

On the number of general BCs identified ("#gen."), Tab uses table structure to easily construct general BCs from semantics, but limited by algorithm performance, its number is not as good as LOGION. GA only searches for BCs without the bias towards general ones, so it cannot find many general BCs. Thanks to the semantic guiding of strengthening and weakening operations in the local search process, LOGION can find more general BCs.

On successfully solved cases ("#suc."), LOGION have more successes, indicating that our algorithm has better robustness. What is more, LOGION is the first approach that can handle AMBA with 2 masters and Genbuf within 5 senders. These industrial cases are so difficult for Tab that we extend the termination condition to 1 day and Tab does not yield any answers.

Considering that Degiovanni et al. (2018b) ran GA for 50 iterations (known as generations of the population), we also show the results, denoted by GA-50Ite, in Table 7. We directly use the experimental results provided Degiovanni et al. (2018b)[6] for artificial cases. For the new industrial cases, we run GA for 50 iterations. Note that as the case size becomes larger, the time for 50 iterations of GA gradually increases, especially for the industrial cases (exceeded one day). In contrast, LOGION still sets a time limit of 1 h. Obviously, for some cases, *e.g.*, LB, LC, and AMBA, GA-50Ite is able to identify BCs, while it cannot identify BCs under the 1 h limit. It suggests that GA-50Ite is much better than GA thanks to a longer running time. However, LOGION is still better than GA-50Ite. For industrial cases, we use GA-50Ite to run for more than one day, and GA-50Ite still does not find BCs.

To answer RQ 2, LOGION is better than GA in terms of the number of BCs, quality, and robustness.

6.5. Effectiveness of local search

In Table 8, comparing LOGION (resp. LOGION-Aa) and LOGION-simp (resp. LOGION-simp-Aa), we observe that LOGION (resp. LOGION-Aa) finds more general BCs while LOGION-simp (resp. LOGION-simp-Aa) finds more BCs. LOGION-simp and LOGION-simp-Aa take a short time in each iteration and performs many iterations in an hour because they does not use the time-consuming strengthening and weakening operators. However, the operators can guid to search for more general BCs, which confirm the advantage of our well-designed local search to answer RQ 3.

6.6. Effectiveness of LISF

Table 9 shows that LOGION-Aa has advantage on smaller cases. When cases are small, the benefit of using pre-checking is not obvious. For the formulae that are hit by pre-checking, the performances of LTL satisfiability checking and trace checking are comparable. For the formulae that are not hit, LISF has additional overhead for trace checking.

As the scale of the cases increases, LOGION and LOGION-nu gradually dominate. LOGION-nu is better than LOGION-Aa, which shows that nuXmv is more suitable for large formulae. As the scale further

[6] https://dc.exa.unrc.edu.ar/staff/rdegiovanni/ASE2018.html.

Table 6
Experimental results compared with the SOTA approaches.

Cases	Tab			GA			LOGION		
	#bc	#gen.	time	#bc	#gen.	#suc.	#bc	#gen.	#suc.
RP1	1.0	0.3	0.1	24.1	0.8	10	**2153.0**	**4.0**	10
RP2	1.0	0.0	0.4	25.4	0.6	10	**1872.4**	**4.5**	10
Ele	1.0	0.0	0.7	25.6	1.6	10	**2467.1**	**11.1**	10
TCP	2.0	0.0	1.0	73.2	0.0	10	**2000.6**	**4.7**	10
AAP	4.0	3.9	0.3	45.6	0.1	10	**1772.5**	**9.3**	10
MP	1.0	0.0	0.0	50.8	0.9	10	**2078.2**	**9.6**	10
ATM	3.0	0.9	1.9	59.7	0.4	10	**2309.0**	**18.8**	10
RRCS	1.0	0.0	0.3	37.2	1.8	10	**1467.4**	**31.3**	10
Tel	1.0	0.0	8.6	40.8	0.7	8	**1867.3**	**31.2**	10
LAS	1.0	0.0	3.2	133.0	0.1	2	**1065.0**	**8.9**	10
PA	N/A	N/A	>1 day	304.0	0.0	1	**947.8**	**13.8**	10
RRA	N/A	N/A	>1 day	N/A	N/A	0	**1400.8**	**55.0**	10
SA	N/A	N/A	>1 day	N/A	N/A	0	**1148.5**	**5.2**	10
LB	N/A	N/A	>1 day	N/A	N/A	0	**789.0**	**20.0**	10
LC	N/A	N/A	>1 day	N/A	N/A	0	**648.1**	**50.2**	10
AMBA	N/A	N/A	>1 day	N/A	N/A	0	**2.5**	**0.9**	6
AMBA02	N/A	N/A	>1 day	N/A	N/A	0	**19.7**	**12.1**	10
Genbuf01	N/A	N/A	>1 day	N/A	N/A	0	**70.6**	**16.6**	10
Genbuf02	N/A	N/A	>1 day	N/A	N/A	0	**45.8**	**12.7**	10
Genbuf03	N/A	N/A	>1 day	N/A	N/A	0	**23.2**	**9.9**	10
Genbuf04	N/A	N/A	>1 day	N/A	N/A	0	**12.6**	**10.1**	10
Genbuf05	N/A	N/A	>1 day	N/A	N/A	0	**7.5**	**6.9**	10

Table 7
Experimental results compared with GA-50Ite.

Cases	GA-50Ite				LOGION		
	#bc	#gen.	Time (s)	#suc.	#bc	#gen.	#suc.
RP1	27.8	3.3	39.3	10	**2153.0**	**4.0**	10
RP2	22.8	2.3	34.5	8	**1872.4**	**4.5**	10
Ele	7.7	2.4	3.6	10	**2467.1**	**11.1**	10
TCP	8.3	2.1	14.6	9	**2000.6**	**4.7**	10
AAP	21.6	2.7	10.4	10	**1772.5**	**9.3**	10
MP	18.0	3.8	9.9	10	**2078.2**	**9.6**	10
ATM	9.0	2.3	11.9	9	**2309.0**	**18.8**	10
RRCS	16.1	3.5	23.4	8	**1467.4**	**31.3**	10
Tel	24.0	3.3	50.6	3	**1867.3**	**31.2**	10
LAS	84.7	4.3	5746.5	3	**1065.0**	**8.9**	10
PA	13.5	2.3	3910.6	4	**947.8**	**13.8**	10
RRA	37.1	0.5	174.1	9	**1400.8**	**55.0**	10
SA	15.3	1.1	1564.9	8	**1148.5**	**5.2**	10
LB	3.0	0.2	8601.4	3	**789.0**	**20.0**	10
LC	3.4	0.6	18 964.4	5	**648.1**	**50.2**	10
AMBA	**2.8**	0.3	12 528.9	6	2.5	**0.9**	6
AMBA02	N/A	N/A	>1 day	0	**19.7**	**12.1**	10
Genbuf01	N/A	N/A	>1 day	0	**70.6**	**16.6**	10
Genbuf02	N/A	N/A	>1 day	0	**45.8**	**12.7**	10
Genbuf03	N/A	N/A	>1 day	0	**23.2**	**9.9**	10
Genbuf04	N/A	N/A	>1 day	0	**12.6**	**10.1**	10
Genbuf05	N/A	N/A	>1 day	0	**7.5**	**6.9**	10

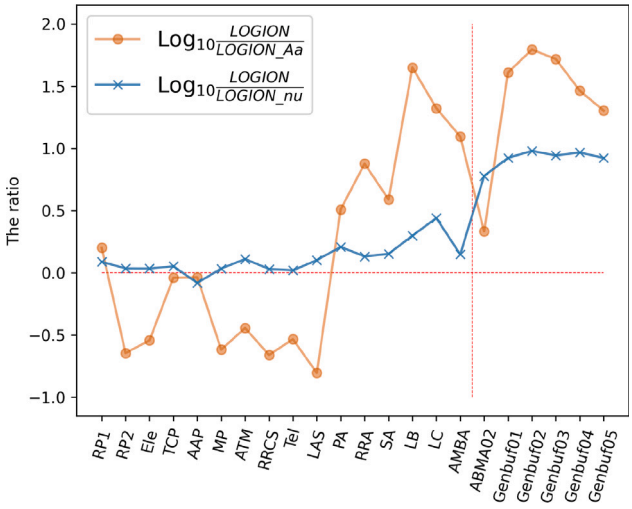

Fig. 2. The comparison of the number of checked candidate BCs of LOGION, LOGION-Aa, and LOGION-nu. The red vertical line is used to separate artificial cases and industrial cases. The Y-axis means their comparison on the number of candidate BCs being checked during the searching.

increases, reaching the industrial level, LOGION is significantly better than LOGION-Aa and LOGION-nu. Note that, in most cases, the hitting percentage of LISF is higher than 85%, except AAP, which shows that LISF filters most of the checked formulae by the lasso set. In other words, for most of the candidate BCs, they are checked in bilinear time. This explains that the advantage of LOGION is that the pre-checking of LISF has a very high hit rate, which can greatly speed up the BC verification.

Fig. 2 demonstrates the number of checked candidate BCs of LOGION, LOGION-Aa, and LOGION-nu. Combining Fig. 2 and Table 9, we observe that if an approach checks more candidates (in relatively large cases), it has better performance. It shows that the performance of identifying BCs has been improved significantly if checking BCs is efficient, which is exactly why LISF works.

To answer RQ 3, thanks to speeding up the BC verification process, LISF significantly improves the performance of identifying BCs, especially on large-scale cases.

7. Related work

Besides the informal or semi-formal approaches to managing inconsistencies, such as Hausmann et al. (2002), Herzig and Paredis (2014), Kamalrudin (2009) and Kamalrudin et al. (2011), a series of formal approaches (Nuseibeh and Russo, 1999; Ellen et al., 2014; Ernst et al., 2012; Harel et al., 2005; Nguyen et al., 2014) recently have been proposed, which only focus on logical inconsistency or ontology mismatch. In this paper, we are interested in identifying the situations that lead to goal divergences, which are weak inconsistencies.

Recently, the conflict analysis phase in GORE has been widely discussed. For the assessment of conflicts, Degiovanni et al. (2018a) recently have proposed an automated approach to assess how likely a conflict is, under an assumption that all events are equally likely. Recently, Luo et al. (2021) proposed the contrasty metric, characterizing the relationship of system divergence captured by BCs, to assess the quality of BCs to help resolve. For the resolution of conflicts, Murukannaiah et al. (2015) resolved the conflicts among stakeholder goals of

Table 8
Experimental results about the effectiveness of local search. "%hit" means the percentage of the number of formulae which are filtered in line 4 of LISF over all calls.

Cases	LOGION-simp-Aa			LOGION-simp				LOGION-Aa			LOGION			
	#bc	#gen.	#suc.	#bc	#gen.	#suc.	%hit	#bc	#gen.	#suc.	#bc	#gen.	#suc.	%hit
RP1	**9 891.7**	0.3	7	1114.0	0.4	5	97.0%	1 170.8	2.5	10	2153.0	**5.1**	10	90.0%
RP2	**19 415.4**	3.3	7	1281.5	0.1	6	95.2%	8 192.1	4.0	10	1872.4	**5.1**	10	90.3%
Ele	4 664.2	20.2	5	2774.5	1.7	6	94.0%	**8 661.4**	**14.0**	10	2467.1	12.2	10	85.4%
TCP	**7 815.8**	0.1	6	1777.1	0.0	7	94.8%	2 016.8	3.9	10	2000.6	**5.5**	10	86.8%
AAP	**14 200.2**	**27.9**	10	2303.8	3.3	8	92.6%	1 520.3	7.8	10	1772.5	12.8	10	70.8%
MP	**27 215.1**	5.2	8	2180.4	4.0	9	93.6%	10 489.4	**25.4**	10	2078.2	10.1	10	88.2%
ATM	**13 894.3**	11.0	8	1509.2	3.4	6	95.2%	4 886.9	5.9	10	2309.0	**18.7**	10	88.7%
RRCS	**17 554.3**	28.8	6	1385.5	4.7	2	96.4%	8 262.8	**39.9**	10	1467.4	32.3	10	91.3%
Tel	**7 513.5**	5.6	4	788.4	8.2	5	95.7%	6 425.0	**42.0**	10	1867.3	34.6	10	89.9%
LAS	N/A	N/A	0	N/A	N/A	0	96.4%	**5 130.4**	9.4	10	1065.0	**11.5**	10	92.5%
PA	146.0	0.7	1	N/A	N/A	0	96.7%	311.1	4.4	10	**947.8**	**15.5**	10	93.2%
RRA	797.5	7.8	2	N/A	N/A	0	96.6%	201.9	11.0	10	**1400.8**	**68.3**	10	92.2%
SA	**3 313.0**	0.5	1	N/A	N/A	0	97.0%	240.1	4.0	10	1148.5	**6.0**	10	92.1%
LB	6.0	0.2	1	N/A	N/A	0	96.5%	N/A	N/A	0	**789.0**	**28.3**	10	94.5%
LC	81.3	24.5	9	N/A	N/A	0	94.1%	13.2	1.0	5	**648.1**	**53.8**	10	93.3%
AMBA	N/A	N/A	0	N/A	N/A	0	94.5%	N/A	N/A	0	**2.5**	**0.9**	6	94.1%
ABMA02	N/A	N/A	0	N/A	N/A	0	85.2%	N/A	N/A	0	**19.7**	**12.1**	10	96.1%
Genbuf01	N/A	N/A	0	N/A	N/A	0	89.4%	N/A	N/A	0	**70.6**	**16.7**	10	95.9%
Genbuf02	N/A	N/A	0	N/A	N/A	0	90.2%	N/A	N/A	0	**45.8**	**12.7**	10	96.5%
Genbuf03	N/A	N/A	0	N/A	N/A	0	86.6%	N/A	N/A	0	**23.2**	**9.9**	10	95.7%
Genbuf04	N/A	N/A	0	N/A	N/A	0	82.5%	N/A	N/A	0	**12.6**	**10.2**	10	95.6%
Genbuf05	N/A	N/A	0	N/A	N/A	0	78.6%	N/A	N/A	0	**7.5**	**6.9**	10	93.7%

Table 9
Experimental results about the effectiveness of LISF.

Cases	LOGION-Aa			LOGION-nu			LOGION			
	#bc	#gen.	#suc.	#bc	#gen.	#suc.	#bc	#gen.	#suc.	%hit
RP1	1 170.8	1.4	10	1767.0	**4.2**	10	**2153.0**	4.0	10	90.0%
RP2	**8 192.1**	3.6	10	1889.7	**4.5**	10	1872.4	**4.5**	10	90.3%
Ele	**8 661.4**	7.3	10	2330.8	**15.9**	10	2467.1	11.1	10	85.4%
TCP	**2 016.8**	3.7	10	1610.7	**4.7**	10	2000.6	**4.7**	10	86.8%
AAP	1 520.3	6.4	10	**1803.3**	**14.1**	10	1772.5	9.3	10	70.8%
MP	**10 489.4**	**25.4**	10	1946.5	13.8	10	2078.2	9.6	10	88.2%
ATM	**4 886.9**	6.5	10	1779.5	**24.5**	10	2309.0	18.8	10	88.7%
RRCS	**8 262.8**	**37.2**	10	1407.6	21.8	10	1467.4	31.3	10	91.3%
Tel	**6 425.0**	**40.9**	10	1803.2	19.6	10	1867.3	31.2	10	89.9%
LAS	**5 130.4**	4.3	10	760.0	7.1	10	1065.0	**8.9**	10	92.5%
PA	311.1	3.0	10	587.0	13.6	10	**947.8**	**13.8**	10	93.2%
RRA	201.9	9.8	10	1083.3	45.0	10	**1400.8**	**55.0**	10	92.2%
SA	240.1	3.2	10	828.4	**6.8**	10	**1148.5**	5.2	10	92.1%
LB	N/A	N/A	0	438.5	12.7	10	**789.0**	**20.0**	10	94.5%
LC	13.2	0.8	5	276.9	30.0	10	**648.1**	**50.2**	10	93.3%
AMBA	N/A	N/A	0	1.5	0.3	2	**2.5**	**0.9**	6	94.1%
AMBA02	N/A	N/A	0	2.2	2.2	10	**19.7**	**12.1**	10	96.1%
Genbuf01	N/A	N/A	0	6.5	6.4	10	**70.6**	**16.6**	10	95.9%
Genbuf02	N/A	N/A	0	3.9	3.6	10	**45.8**	**12.7**	10	96.5%
Genbuf03	N/A	N/A	0	2.0	1.4	7	**23.2**	**9.9**	10	95.7%
Genbuf04	N/A	N/A	0	1.5	1.1	8	**12.6**	**10.1**	10	95.6%
Genbuf05	N/A	N/A	0	1.0	0.6	6	**7.5**	**6.9**	10	93.7%

system-to-be based on the Analysis of Competing Hypotheses technique and argumentation patterns. Related works on conflict resolution also include Felfernig et al. (2009) which calculates the personalized repairs for the conflicts of requirements with the principle of model-based diagnosis. These approaches assume that conflicts are already identified, and our approach lays the groundwork for addressing these problems.

Let us come back to the goal-conflict identification problem. Existing approaches mainly categorize into construct-based approaches and search-based approaches. For construct-based approaches, Van Lamsweerde et al. (1998) proposed a pattern-based approach which only returns a BC in a pre-defined limited form. While LOGION has no limitation on the specifications and is able to generate BCs in any form theoretically. Degiovanni et al. (2016) exploited a tableaux-based approach that generates general BCs. However, their approach relies on sophisticated logical mechanisms that handle specific semantic structures derived from goals and domain characteristics. Besides, they face scalability issues due to the scalability limitations of the underlying logical mechanisms, rendering them suitable only for small-scale specifications. As shown in Section 6, LOGION identifies significantly more BCs, even for large cases. For the search-based approach, Degiovanni et al. (2018b) presented a genetic algorithm which just seeks for BCs and handles specifications that are beyond the scope of previous approaches. LOGION is also a search-based approach that has an advantage over their approach in terms of efficiency because it employs a local search algorithm, taking full advantage of the structural similarity of BCs, and efficient LISF. Experimental results confirm their effectiveness.

8. Conclusion

Goal-conflict identification is an important stage of GORE. In this paper, we have proposed an efficient local search algorithm LOGION which combines the syntax and semantics manipulation of LTL formulae to identify general BCs. Besides, we have designed LISF to greatly optimize the BC verification. The experimental results show that

LOGION extremely outperforms the SOTA and handles more large-scale cases.

CRediT authorship contribution statement

Weilin Luo: Conceptualization, Methodology, Writing, Review & Editing, Funding acquisition. **Polong Chen:** Methodology, Investigation, Data curation, One of the principal proposers of the basic approach proposed in this paper, Collaborated with Weilin Luo to complete the revision of this work from the conference version to the journal version, Independently completed all supplementary experiments in the response stage. **Hai Wan:** Conceptualization, Methodology, Supervision, Funding acquisition. **Hongzhen Zhong:** Formal analysis, Project administration, Software, Writing – original draft. **Shaowei Cai:** Conceptualization. **Zhanhao Xiao:** Writing – original draft, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

We thank Jianwen Li from East China Normal University for discussion on the paper and anonymous referees for helpful comments. This paper was supported by the National Natural Science Foundation of China (No. 62276284, 61976232, 51978675 and 61906216), Guangdong Basic and Applied Basic Research Foundation (No. 2023A1515011470, 2022A1515011355 and 2022A1515012429), Guangzhou Science and Technology Project (No. 202201011699 and 2024A04J4676), Guizhou Provincial Science and Technology Projects (No. 2022-259), Humanities and Social Science Research Project of Ministry of Education (18YJCZH006), as well as Fundamental Research Funds for the Central Universities, Sun Yat-sen University (No. 23ptpy31).

References

Alrajeh, D., Kramer, J., Russo, A., Uchitel, S., 2009. Learning operational requirements from goal models. In: ICSE. pp. 265–275.
Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M., 2007. Automatic hardware synthesis from specifications: A case study. In: DATE. pp. 1–6.
Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S., 2014. The nuxmv symbolic model checker. In: CAV. pp. 334–342.
De Beukelaer, H., Davenport, G.F., De Meyer, G., Fack, V., 2017. JAMES: An object-oriented java framework for discrete optimization using local search metaheuristics. Softw. - Pract. Exp. 47 (6), 921–938.
Degiovanni, R., Alrajeh, D., Aguirre, N., Uchitel, S., 2014. Automated goal operationalisation based on interpolation and SAT solving. In: ICSE. pp. 129–139.
Degiovanni, R., Castro, P., Arroyo, M., Ruiz, M., Aguirre, N., Frias, M., 2018a. Goal-conflict likelihood assessment based on model counting. In: ICSE. pp. 1125–1135.
Degiovanni, R., Molina, F., Regis, G., Aguirre, N., 2018b. A genetic algorithm for goal-conflict identification. In: ASE. pp. 520–531.
Degiovanni, R., Ricci, N., Alrajeh, D., Castro, P., Aguirre, N., 2016. Goal-conflict detection based on temporal satisfiability checking. In: ASE. pp. 507–518.
Ellen, C., Sieverding, S., Hungar, H., 2014. Detecting consistencies and inconsistencies of pattern-based functional requirements. In: FMICS. pp. 155–169.
Enderton, H., Enderton, H.B., 2001. A Mathematical Introduction to Logic. Elsevier.
Ernst, N.A., Borgida, A., Mylopoulos, J., Jureta, I.J., 2012. Agile requirements evolution via paraconsistent reasoning. In: CAiSE. pp. 382–397.
Fahmideh, M., Beydoun, G., 2018. Reusing empirical knowledge during cloud computing adoption. J. Syst. Softw. 138, 124–157.
Felfernig, A., Friedrich, G., Schubert, M., Mandl, M., Mairitsch, M., Teppan, E., 2009. Plausible repairs for inconsistent requirements. In: IJCAI. pp. 791–796.
Fisman, D., Kupferman, O., Sheinvald-Faragy, S., Vardi, M.Y., 2008. A framework for inherent vacuity. In: HVC. pp. 7–22.
Giannakopoulou, D., Lerda, F., 2002. From states to transitions: Improving translation of LTL formulae to Büchi automata. In: FORTE. pp. 308–326.
Glover, F., 1986. Future paths for integer programming and links to artificial intelligence. Comput. OR 13 (5), 533–549.
Hansen, P., Jaumard, B., 1990. Algorithms for the maximum satisfiability problem. ACM J. Exp. Algorithmics 44 (4), 279–303.
Harel, D., Kugler, H., Pnueli, A., 2005. Synthesis revisited: Generating statechart models from scenario-based requirements. In: Formal Methods in Software and Systems Modeling. Springer, pp. 309–324. http://dx.doi.org/10.1007/978-3-540-31847-7_18.
Hausmann, J.H., Heckel, R., Taentzer, G., 2002. Detection of conflicting functional requirements in a use case-driven approach. In: ICSE. pp. 105–115.
Herzig, S.J., Paredis, C.J., 2014. A conceptual basis for inconsistency management in model-based systems engineering. Proc. CIRP 21, 52–57.
Hoos, H.H., Stützle, T., 2004. Stochastic Local Search: Foundations and Applications. Elsevier.
Kamalrudin, M., 2009. Automated software tool support for checking the inconsistency of requirements. In: ASE. pp. 693–697.
Kamalrudin, M., Hosking, J., Grundy, J., 2011. Improving requirements quality using essential use case interaction patterns. In: ICSE. pp. 531–540.
Kramer, J., Magee, J., Sloman, M., Lister, A., 1983. CONIC: an integrated approach to distributed computer control systems. IET Comput. Digit. Tech. 130 (1), 1–10.
Li, J., Zhu, S., Pu, G., Vardi, M.Y., 2015. SAT-based explicit LTL reasoning. In: HVC. pp. 209–224.
Li, J., Zhu, S., Pu, G., Zhang, L., Vardi, M.Y., 2019. SAT-based explicit LTL reasoning and its application to satisfiability checking. Form. Methods Syst. Des. 54 (2), 164–190.
Ltd, A., 1999. AMBA specification (Rev. 2). Available from www.arm.com.
Luo, W., Wan, H., Song, X., Yang, B., Zhong, H., Chen, Y., 2021. How to identify boundary conditions with contrasty metric? In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. pp. 1473–1484.
Markey, N., Schnoebelen, P., 2003. Model checking a path. In: CONCUR. pp. 251–265.
Maté, A., Trujillo, J., Franch, X., 2014. Adding semantic modules to improve goal-oriented analysis of data warehouses using I-star. J. Syst. Softw. 88, 102–111.
Murukannaiah, P.K., Kalia, A.K., Telangy, P.R., Singh, M.P., 2015. Resolving goal conflicts via argumentation-based analysis of competing hypotheses. In: RE. pp. 156–165.
Nguyen, T.H., Vo, B.Q., Lumpe, M., Grundy, J., 2014. KBRE: a framework for knowledge-based requirements engineering. Softw. Qual. J. 22 (1), 87–119.
Nuseibeh, B., Russo, A., 1999. Using abduction to evolve inconsistent requirements specification. Australas. J. Inf. Syst. 7 (1; SPI), 118–130.
Pnueli, A., 1977. The temporal logic of programs. In: FOCS. pp. 46–57.
Rico-Juan, J.R., Micó, L., 2003. Comparison of AESA and LAESA search algorithms using string and tree-edit-distances. Pattern Recognit. Lett. 24 (9–10), 1417–1426.
Sistla, A.P., Clarke, E.M., 1985. The complexity of propositional linear temporal logics. J. ACM 32 (3), 733–749.
Van Lamsweerde, A., 2009. Requirements Engineering: From System Goals to UML Models to Software, vol. 10, John Wiley & Sons, Chichester, UK.
Van Lamsweerde, A., Darimont, R., Letier, E., 1998. Managing conflicts in goal-driven requirements engineering. IEEE Trans. Softw. Eng. 24 (11), 908–926.
Van Lamsweerde, A., Letier, E., 1998. Integrating obstacles in goal-driven requirements engineering. In: ICSE. pp. 53–62.
Zhang, K., Shasha, D., 1989. Simple fast algorithms for the editing distance between trees and related problems. SIAM J. Comput. 18 (6), 1245–1262.
Zhong, H., Wan, H., Luo, W., Xiao, Z., Li, J., Fang, B., 2020. Structural similarity of boundary conditions and an efficient local search algorithm for goal conflict identification. In: 27th Asia-Pacific Software Engineering Conference, APSEC 2020, Singapore, December 1-4, 2020. pp. 286–295.

Weilin Luo received the Ph.D. degree from Sun Yat-sen University, China, in 2022. He is currently an associate research fellow at School of Computer Science and Engineering, Sun Yat-sen University, China. His research interests include reliability analysis, formal methods, software engineering, and artificial intelligence. Currently, he is working on neuro-symbolic computing and its applications in software engineering. He has published many papers as the first author in IEEE Transactions on Reliability, ICSE, ASE, AAAI, IJCAI, ICCAD, etc.