Structural Similarity of Boundary Conditions and an Efficient Local Search Algorithm for Goal Conflict Identification

1st Hongzhen Zhong
School of Data and Computer Science
Sun Yat-sen University
Guangzhou, China
zhonghzh5@mail2.sysu.edu.cn

2nd Hai Wan†
School of Data and Computer Science
Sun Yat-sen University
Guangzhou, China
wanhai@mail.sysu.edu.cn

3rd Weilin Luo
School of Data and Computer Science
Sun Yat-sen University
Guangzhou, China
luowlin3@mail2.sysu.edu.cn

4th Zhanhao Xiao
School of Data and Computer Science
Sun Yat-sen University
Guangzhou, China
xiaozhh9@mail.sysu.edu.cn

5th Jia Li
School of Data and Computer Science
Sun Yat-sen University
Guangzhou, China
lijia49@mail2.sysu.edu.cn

6th Biqing Fang
School of Data and Computer Science
Sun Yat-sen University
Guangzhou, China
fangbq3@mail3.sysu.edu.cn

2020 27th Asia-Pacific Software Engineering Conference (APSEC) | 978-1-7281-9553-7/20/$31.00 ©2020 IEEE | DOI: 10.1109/APSEC51365.2020.00037

***Abstract*—In goal-oriented requirements engineering, goal conflict identification is of fundamental importance for requirements analysis. The task aims to find the feasible situations which make the goals diverge within the domain, called *boundary conditions (BCs)*. However, the existing approaches for goal conflict identification fail to find sufficient BCs and general BCs which cover more combinations of circumstances. From the BCs found by these existing approaches, we have observed an interesting phenomenon that there are some pairs of BCs are similar in formula structure, which occurs frequently in the experimental cases. In other words, once a BC is found, a new BC may be discovered quickly by slightly changing the former. It inspires us to develop a local search algorithm named `LOGION` to find BCs, in which the structural similarity is captured by the neighborhood relation of formulae. Based on structural similarity, `LOGION` can find a lot of BCs in a short time. Moreover, due to the large number of BCs identified, it potentially selects more general BCs from them. By taking experiments on a set of cases, we show that `LOGION` effectively exploits the structural similarity of BCs. We also compare our algorithm against the two state-of-the-art approaches. The experimental results show that `LOGION` produces one order of magnitude more BCs than the state-of-the-art approaches and confirm that `LOGION` finds out more general BCs thanks to a large number of BCs.**

***Index Terms*—Goal Conflicts, Boundary Condition, Local Search, LTL Satisfiability**

I. Introduction

Requirements engineering is an essential phase of the software development life cycle [1]. It is important to attain the correct software requirements specification. Many researches have demonstrated the significant advantages that formal and goal-oriented approaches bring to the generation of correct specifications [2]–[4]. In such approaches, *domain properties* and *goals*, represented in the *Linear-time Temporal Logic (LTL)*, can capture requirements naturally [5]. *Goal conflict identification* is an important stage of formal and goal-oriented requirements engineering, which aims to find inconsistencies between goals. The inconsistencies are captured by *boundary conditions (BCs)* under which goals are unsatisfiable as a whole within the domain properties. Below we illustrate BCs using a MinePump example [6].

†Corresponding author

Example 1. *Consider a system to control a pump inside a mine. The main goal of the system is to avoid flood in the mine. The system has two sensors. One detects the high water level (h), the other detects methane in the environment (m). When the water level is high, the system should turn on the pump (p). When there is methane in the environment, the pump should be turned off. 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**: $\square((p \wedge \bigcirc p) \rightarrow \bigcirc(\bigcirc\neg h))$*

Goals:

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

2640-0715/20/$31.00 ©2020 IEEE
DOI 10.1109/APSEC51365.2020.00037

Two of BCs of this specification are:

$$BC_1 = \Box(h \wedge m)$$
$$BC_2 = \Diamond(h \wedge m)$$

Intuitively, BC_1 captures the situation that high water level and methane always happen, while BC_2 captures the situation that high water and methane will happen in the future. Under the BC_1 or BC_2, two goals are unsatisfiable simultaneously within domain property.

Van Lamsweerde [5] pointed out that we must identify as many BCs as possible, which is important to refine requirements. Identifying more BCs helps us select more general BCs. Intuitively, the more general BC is of higher quality because it can cover more combinations of circumstances. Once it is worked out, less general BCs will also be worked out. In order to find more BCs, previous approaches can be mainly categorized into construct-based approaches and search-based approaches. Construct-based approaches focus on generating BCs based on constructing templates [7] or tableau structure [8]. However, they suffer from scalability issues because the templates and the tableau structure could be generated only in a small requirement specification. The search-based approach [9] focuses on finding BCs. It produces lots of LTL formulae and checks whether they are BCs one by one. But it searches without guided and wastes a lot of time checking the formulae that are not BCs.

When we fixed our eyes on the BCs computed by the existing approaches to generating BCs [8], [9], we found an interesting phenomenon that there exist some pairs of BCs which are similar on the structure. Recall that BC_1 and BC_2 are presented in Example 1, these two formulae are similar on the structure: they differ from each other in only one place. Furthermore, BC_2 is more general than BC_1 because BC_1 is just a special case of BC_2. Based on structural similarity, we can find more general BC_2, not just BC_1. We will give a formal definition of general BC in section II-A. In consequence, once one BC is found, the other one may be found via simply changing the former. Fortunately, we have observed that this case occurs frequently in the specifications, which allows us to find more BCs quickly and help us to select more general BCs via taking advantage of such a kind of structural similarity. To formalize the sightly change of formulae, we propose three formula edit operations: Rename, Insert, and Delete. For that, we introduce an efficient local search based algorithm to compute BCs, in which the structural similarity is captured by the neighborhood relation of formulae. A formula differs from its neighbor on only one or two variables or operators.

In this paper, we evaluate to what extent our approach LOGION exploits the structural similarity of BCs on a set of classical cases. The experimental results show that the BCs successively computed by LOGION are quite similar on the formula structure.

We also compare our approach LOGION against Tableaux-based [8] and the GA-based [9] approaches. The experiment results show that our approach LOGION finds one order of magnitude more BCs than these two approaches in almost all cases. We also choose more general BCs based on the large number of BCs found by LOGION.

Our main contributions are summarized as follows.

1) From the BCs found by the Tableaux-based [8] and the GA-based [9] approaches, we discover an interesting phenomenon that there are some pairs of BCs are similar in formula structure.
2) We propose an efficient local search algorithm named LOGION for goal-conflict identification, in which the neighborhood of formula captures the structural similarity of BCs.
3) Empirical evidence shows that our approach LOGION has superiority on the efficiency of computing BCs, over the Tableaux-based [8] and the GA-based [9] approaches.

The remainder of the paper is organized as follows. In Section II, we give preliminaries about goal-conflict identification, linear-time temporal logic, tree edit distance, and local search algorithms. Next, we show how the similarity of BCs occurs frequently in Section III and describe our approach in detail in Section IV. In Section V, we carry out our experiments to validate our approach and compare it with related approaches. Finally, we discuss related work in Section VI and make some conclusions in the last section.

II. Background

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

A. Goal Conflict Identification

In goal-oriented requirements engineering methodologies [5], *goals* and *domain properties* are formed in linear-time temporal logic. Goals are prescriptive statements that the system must achieve, while domain properties are descriptive statements that capture the domain about the problem world. It is unrealistic to require requirements specification to be complete or all goals to be satisfiable ideally, because unanticipated cases may occur. It suggests identifying the conflicts as early as possible. The goal-conflict identification stage is important in the conflict analysis phase.

The *conflict analysis phase* [5], [10] has three main stages:

1) the *identification stage* is to identify condition whose occurrence makes the goals diverge;
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 the identification stage. A conflict represents a condition that occurrence results in the loss of satisfaction of the goals.

Definition 1. *Given a set $\{G_1, \ldots, G_n\}$ of goals and a set Dom of domain properties, the goals are said to be divergent in the context of φ if there exists an expression φ, called a* boundary condition*, such that the following properties hold:*

$$Dom \wedge G \wedge \varphi \models \bot \quad \text{(logical inconsistency)}$$
$$Dom \wedge G_{-i} \wedge \varphi \not\models \bot, \textit{ for each } 1 \leq i \leq n \quad \text{(minimality)}$$
$$\neg G \not\equiv \varphi \quad \text{(non-triviality)}$$

where $G = \bigwedge_{1 \leq i \leq n} G_i$ and $G_{-i} = \bigwedge_{j \neq i} G_j$.

Intuitively, a BC captures a situation where the goals as a whole are not satisfiable. The logical inconsistency condition means that the conjunction of goals $G_1, \ldots, G_n$ becomes inconsistent when BC holds. The minimality condition state that disregarding one of the goals no longer results in the consistency. The non-triviality condition forbids a BC to be a trivial condition, which is the negation of the conjunction of the goals. Note that the minimality condition requires a BC itself to be consistent.

Specifying software requirements in the LTL formulation allows us to employ automated LTL satisfiability (SAT) solvers to check for the feasibility of the corresponding requirements. With an efficient LTL SAT solvers, we can automatically check if the generated candidate formulae are valid BCs or not by checking if they satisfy the properties.

Given a set of BC S, we call BC $\varphi_i \in S$ is more general than another BC $\varphi_j \in S$ if φ_j implies φ_i. 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.

B. Linear-Time Temporal Logic

Linear-Time Temporal Logic (LTL) [11] is widely used to describe infinite behaviors of discrete systems. LTL formulae are defined from a countably infinite set $\mathbb{P}$ of propositional variables, classical propositional connectives including $\neg$, $\wedge$, $\vee$ and temporal operators $\Box$ (always), $\Diamond$ (eventually), $\bigcirc$ (next), $\mathcal{U}$ (until), $\mathcal{R}$ (release) and $\mathcal{W}$ (weak-until), as follows:

1) $b \in \mathbb{B}$ is an LTL formula, where $\mathbb{B} = \{\top, \bot\}$;
2) every proposition $p \in \mathbb{P}$ is an LTL formula;
3) if φ_1 and φ_2 are LTL formulae, then so are $\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\bigcirc\varphi_1, \Box\varphi_1, \Diamond\varphi_1$, $\varphi_1 \mathcal{U} \varphi_2, \varphi_1 \mathcal{R} \varphi_2, \varphi_1 \mathcal{W} \varphi_2$.

LTL formulae are interpreted over linear-time structures. A linear-time structure is a pair of $M = (S, \varepsilon)$ where S is an ω-sequence $s_0, s_1, \ldots$ of states and $\varepsilon : S \to 2^{\mathbb{P}}$ is a function mapping each state s_i to a set of propositional variables that hold in s_i. Let M be a linear-time structure, $i \in \mathbb{N}^0$ a position, and φ, ψ LTL formulae. We define the satisfaction relation $\models$ as follows:

$$\begin{array}{lll} M, i \models p & \text{iff} & p \in \varepsilon(s_i), \text{ where } p \in \mathbb{P} \\ M, i \models \neg\varphi & \text{iff} & M, i \not\models \varphi \\ M, i \models \varphi \wedge \psi & \text{iff} & M, i \models \varphi \text{ and } M, i \models \psi \\ M, i \models \bigcirc\varphi & \text{iff} & M, i+1 \models \varphi \\ M, i \models \varphi \,\mathcal{U}\, \psi & \text{iff} & \exists k \geq i \text{ s.t. } M, k \models \psi \text{ and} \\ & & \forall i \leq j < k, M, j \models \varphi \end{array}$$

Operator release ($\mathcal{R}$), eventually ($\Diamond$), always ($\Box$), and weak-until ($\mathcal{W}$) are commonly used, and can be defined 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.

An LTL formula φ is satisfiable if there exists a linear-time structure M such that $M, 0 \models \varphi$, and it is valid if $M, 0 \models \varphi$ for all linear-time structures M. The LTL satisfiability and validity problems are decidable and PSPACE-complete [12].

An LTL formula φ implies an LTL formula φ', in notation $\varphi \models \varphi'$, when $M, 0 \models \varphi$ for every linear-time structure M such that $M, 0 \models \varphi'$. Two LTL formulae φ_1 and φ_2 are said to be logically equivalent, denoted by $\varphi_1 \equiv \varphi_2$, if $\varphi_1 \models \varphi_2$ and $\varphi_2 \models \varphi_1$.

An LTL formula ψ is a subformula of an LTL formula φ if ψ is a part of φ. We use $|\varphi|$ to denote the size of the formula φ, *i.e.*, the number of variables and operators in φ.

We refer the reader to [13] for further details on LTL.

C. Tree Edit Distance

The tree edit distance [14] is used to measure the similarity between two ordered labeled trees and has successfully been applied in a wide range of applications. The *tree edit distance* is the cost of the minimal-cost sequence of node edit operations that transforms one tree into another. For ordered label trees, there are three node edit operations:

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

Generally, the cost of each node edit operation is one, and the cost of a sequence is the sum of the cost of its node edit operations. So, the tree edit distance is considered as the length of the sequence with the minimal cost. We use $|T|$ to denote the size of the tree T, *i.e.*, the number of nodes in T. 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 [15], which is defined as:

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

For further details on the tree edit distance, we refer the reader to [16].

D. Local Search

Local search is a kind of meta-heuristic search algorithm. More 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 $N : S(\pi) \mapsto 2^{S(\pi)}$ maps each candidate solution to its neighbors. The set $N(s)$ is called

TABLE I
THE DETAILS OF EACH CASE INCLUDE THE NUMBERS OF DOMAIN PROPERTIES (#DOM), GOALS (#GOAL), VARIABLES (#VAR), AND THE TOTAL SIZE OF ALL FORMULAE (SIZE) FOR THE SPECIFICATION OF EACH CASE

Case	#Dom	#Goal	#Var	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
RRCS	2	2	5	22
Telephone (Tel)	3	2	4	31
LAS	0	5	7	32
Prioritized Arbiter (PA)	6	1	6	57
Round Robin Arbiter (RRA)	6	3	4	77
Simple Arbiter (SA)	5	3	6	84
Load Balancer (LB)	3	7	5	85
LiftController (LC)	7	8	6	124
AMBA	6	21	16	415

the neighbors of s. The objective function $f : S(\pi) \mapsto \mathbb{R}$ 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 cycling, that is, some candidate solutions of high quality are being frequently revisited. *Tabusearch* [17], [18] is a fundamentally different approach to reduce cycling which forbids steps to recently visited candidate solutions. The simplest and most widely applied implementation of tabu search consists of an iterative improvement algorithm enhanced with a form of short-term memory M which stores the last T visited solutions, where T is called the *tabu tenure*. In each search step, the algorithm chooses the best neighbor in $N(s)\backslash M$, where s is the current candidate solution.

For further details on the local search algorithm, we refer the reader to [19].

III. SIMILARITY OF BOUNDARY CONDITIONS

As we show in the introduction section, we have observed that there are some pairs of BCs similar on the structure. In this section, we show the structural similarity of BCs in the classical cases.

A. Cases

We use 16 cases introduced by Degiovanni [9]. Table I summarizes the numbers of domain properties ("#Dom"), goals ("#Goal"), variables ("#Var"), and the total size of all formulae ("Size") for the specification of each case study.

B. Structural Similarity

In order to capture the structural similarity of LTL formulae, we 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 [20]. Then 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.*, $|T_\varphi| = |\varphi|$. The parse tree of the formula $\square(h \rightarrow \bigcirc(p))$ is shown in Figure 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 $\delta(\varphi, \varphi') = \delta(T_\varphi, T_{\varphi'})$ 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 $\Delta(\varphi, \varphi') = \Delta(T_\varphi, T_{\varphi'})$ 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:

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

Example 2 (Example 1 cont.)**.** *Let* $\varphi_1 = \square(h \rightarrow \bigcirc(p))$, $\varphi_2 = \square(h \wedge m)$ *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* $\Delta(\varphi_2, \varphi_3) = 0.125$.

According to the similarity notion of LTL formulae, here we have our first research question:

RQ1: How frequently similar BC pairs do occur in these cases?

As there is no set of BCs for these specifications, we generated BCs by applying the existing approaches [8], [9] for 24 hours (10 runs in parallel). After removing the identical BCs, the numbers of BCs of different cases are shown in Table II, denoted by "#total BC".

In order to explore how frequently similar BC pairs occur, for each case study, 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, which is defined as:

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

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 boundary condition BC, we use $sim(BC, l) = \{BC' | \delta(BC, BC') \leq l\}$ to denote the set of BCs whose formula distance to BC is not greater than l. We use $\#sim(\delta \leq l)$ to denote the average number of similar BCs, which is defined as:

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

The results are shown in Table II. Taking the example of LAS, for each boundary condition BC, there are averagely 6 BCs which differ from BC only on one or two variables or operators. From the table, we can observe that in most cases except for Load Balancer and LiftController, there are more than 70% BCs which can be obtained by applying two or three formula edit operations from another BC. In these cases, every BC averagely has more than one BCs with at most variables or operators differing.

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 II. For example, in RRCS, every boundary condition BC averagely has 6.6 BCs which has a normalized formula distance to BC less than 0.2. The results illustrate that in each case, there are more than 90% BCs having a relatively similar BC whose normalized formula distance is at most 0.3. The results also show that each BC averagely has more than two similar BC whose normalized formula distance is at most 0.2.

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

To sum up, in most cases, there are more than 70% BCs which have both an absolutely similar BC ($\delta \leq 3$) a relatively similar BC ($\Delta \leq 0.3$). Similar BC pairs are very common in cases.

IV. The LOGION Algorithm

In this section, we develop a local search algorithm named LOGION for goal conflict identification, which exploits the structural similarity of BCs to design the neighborhood relation. Next, we specify each component of our algorithm LOGION and then give its entire description.

A. Neighborhood

The search space of LOGION is composed of LTL formulae. To search for BCs, we define the syntactic neighborhood of an LTL formula. Inspire by node edit operations, we introduce LTL *formula edit operations*, which can slightly modify an LTL formula to another LTL formula.

Let o_1, o_1' be two different LTL unary operators (*i.e.*, $o_1, o_1' \in \{\neg, \Box, \Diamond, \bigcirc\}$). and o_2, o_2' be two different LTL binary operators (*i.e.*, $o_2, o_2' \in \{\wedge, \vee, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$). Given a formula ψ, its neighbor ψ' can be constructed by one of three *formula edit operations* defined below:

1) Rename
   a) $\psi' = p'$, when $\psi = p$, $p \in \mathbb{P} \cup \mathbb{B}$, $p' \in \mathbb{P} \cup \mathbb{B}$ and $p \neq p'$
   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 $\psi = \psi_1\ o_2\ \psi_2$

The Rename operation replaces the top symbol of ψ with other same type symbol. The Insert adds a new LTL operator to be top operator of ψ. Note that if the new LTL operator is binary operator, it need to add a new proposition p for creating the valid formula ψ'. The Delete removes the top operator of ψ. If the top operator of ψ is a binary operator. one of the children is a proposition. Then, the Delete removes the top operator and the proposition.

Theorem 1. *For any formulae ψ_1 and ψ_2, ψ_1 can be changed to ψ_2 by a limited number of formula edit operations.*

Proof. We can change ψ_1 to a proposition by Delete and Rename. Then we change the proposition to ψ_2 by Insert and Rename. □

Based on the formula edit operation, we design the neighborhood of an LTL formula.

Definition 2. *Given an LTL formula φ as input, the neighborhood function w.r.t. φ, denoted by $N(\varphi)$, returns all formulae obtained by applying a formula editing operation to a subformulae of φ.*

Specifically, we first extract all subformulae of φ, then we modify each subformula with the above formula edit operations to get $N(\varphi)$.

Property 1. *For an LTL formula φ, if $\varphi' \in N(\varphi)$, then $\delta(\varphi', \varphi) = 1$ or 2.*

Proof. As φ' is a neighbor of φ, φ' is obtained from φ via a formula edit operation. Then we consider the formula edit operation in three cases:

1) In the case of Rename operation, *i.e.*, (1a)-(1c), $T_{\varphi'}$ is obtained from T_φ by relabeling the label of the corresponding node. So $\delta(\varphi', \varphi) = 1$.
2) In the case of unary Insert operation (2a) or unary Delete operation (3a), $T_{\varphi'}$ is obtained by inserted/deleted the corresponding node. So $\delta(\varphi', \varphi) = 1$.
3) In the case of binary Insert operation (2b) or binary Delete operation (3b), $T_{\varphi'}$ is obtained from T_φ by inserting or deleting two corresponding nodes. So $\delta(\varphi', \varphi) = 2$.

□

Property 1 shows that the structural similarity of formulae, as all the neighbors of a formula are formulae whose distance to it is at most 2.

Since there are too many neighbors for φ (such as the $\Box(h \wedge m)$ of Example 1, the number of all its neighbors is 90), to efficiently select a good candidate BC, we bound the number of neighbors to k in $N(\varphi)$, denoted by $N_k(\varphi)$, where k is a hyperparameter. Specifically, we first select k sub-formulae of

TABLE II
EXPERIMENTAL RESULTS ON THE SIMILARITY OF BCS COMPUTED BY THE GA-BASED APPROACH AND TABLEAUX-BASED APPROACH (TABLEAUX FOR SHORT)

Case	GA-based Approach and Tableaux													
	$\%BC(\delta \leq l)$			$\#\mathsf{sim}(\delta \leq l)$			$\%BC(\Delta \leq k)$			$\#\mathsf{sim}(\Delta \leq k)$			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	85.3%	96.4%	98.0%	2.1	6.8	13.4	91.9%	97.1%	98.7%	5.1	21.0	42.3	1.1	308
RP2	84.2%	96.6%	98.8%	2.1	6.8	13.5	87.6%	97.2%	99.1%	8.1	39.4	61.0	1.1	566
Ele	65.9%	93.5%	97.6%	1.2	4.2	9.3	35.8%	90.2%	97.6%	1.0	4.8	11.6	1.2	124
TCP	69.3%	91.9%	98.7%	1.5	5.0	11.3	52.8%	93.2%	98.2%	1.7	6.9	16.2	1.1	382
AAP	63.2%	92.0%	97.9%	1.4	5.0	11.7	24.7%	85.8%	95.1%	0.5	3.4	7.3	1.2	289
MP	68.4%	93.4%	98.2%	1.5	5.4	12.7	54.2%	90.8%	96.8%	1.9	10.0	22.0	1.1	913
ATM	72.6%	94.6%	98.5%	1.7	5.8	12.0	77.9%	97.2%	98.5%	3.6	17.5	37.2	1.1	681
RRCS	40.8%	81.0%	90.2%	0.7	2.3	4.3	66.5%	93.4%	98.1%	1.6	6.6	14.7	1.5	317
Tel	39.2%	72.0%	82.9%	0.7	1.9	3.7	80.5%	95.1%	97.9%	3.1	10.5	23.9	1.7	534
LAS	80.4%	92.5%	99.1%	2.0	6.0	8.9	95.3%	97.2%	97.2%	18.0	24.4	27.4	1.1	108
PA	58.8%	79.1%	87.8%	1.1	3.0	5.6	74.7%	86.9%	92.8%	2.5	7.3	15.5	1.5	321
RRA	86.2%	95.6%	97.8%	2.1	7.0	15.1	92.5%	98.1%	99.4%	4.0	16.8	36.6	1.1	319
SA	29.4%	64.7%	76.5%	0.5	1.1	1.5	64.7%	82.4%	88.2%	2.1	3.8	7.1	1.8	18
LB	7.1%	29.8%	41.7%	0.2	0.5	0.7	63.1%	84.5%	91.7%	0.8	2.1	3.1	3.6	85
LC	0.9%	12.4%	27.8%	0.1	0.3	0.6	36.9%	66.5%	81.7%	0.9	3.4	6.8	4.2	443
AMBA	1.8%	43.3%	72.0%	0.1	1.1	2.0	38.4%	76.8%	91.5%	1.1	6.9	12.5	1.9	165

"$\%BC(\delta \leq l)$" and "$\%BC(\Delta \leq k)$" 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. "$\#\mathsf{sim}(\delta \leq l)$" and "$\#\mathsf{sim}(\Delta \leq l)$" 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 found by the GA-based approach and Tableaux-based approach for 24 hours (10 runs in parallel).

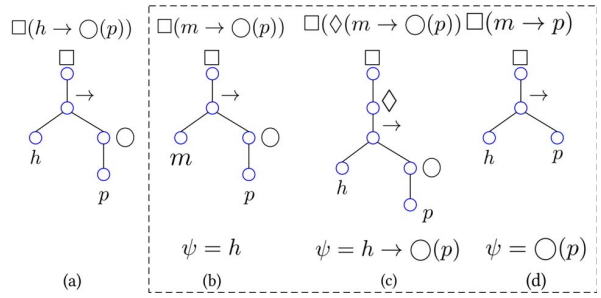

Fig. 1. An example of getting neighbors of $\Box(h \to \bigcirc(p))$ by formula editing operations. (a) $\Box(h \to \bigcirc(p))$ and its parse tree. (b) The neighbor formula after renaming h to m and its parse tree. (c) The neighbor formula after inserting $\Diamond$ and its parse tree. (d) The neighbor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

φ uniformly at random. Then, for each sub-formulae, select a formula edit operation uniformly at random to modify it.

Example 3 (Example 1 cont.)**.** *Considering the formula* $\Box(h \to \bigcirc(p))$ *and* $k = 3$*, suppose we select the subformula* ψ *of* φ*, i.e.,* h*,* $h \to \bigcirc(p)$ *and* $\bigcirc(p)$*. Figure 1(b), (c) and (d) show the result of* Rename *,* Insert *and* Delete *, respectively.*

In order to avoid that some formulae are frequently visited, we employ the tabu search strategy [17], [18]. An LTL formula is called a T-tabu formula iff it has been visited during the last T search steps, where T is called tabu tenure and is a hyperparameter. For an LTL formula φ, its neighborhood *w.r.t.* a T-tabu is defined as its neighbors which are not T-tabu formulae. formally, $N_k(\varphi, T) = \{\beta \mid \beta \in N_k(\varphi)$ and β is not a T-tabu formula$\}$.

B. Initialization and Objective Function

In order to find BCs quickly, the initial formula should be as "close" to a BC as possible. Therefore, we propose $\neg(G_1 \wedge \cdots \wedge G_n)$, called *trivial condition*, as the initial formula. It is easily constructed and "close" to a BC because it only violates non-triviality in Section II.

Example 4 (Example 1 cont.)**.** *Consider the Example 1, the initial formula is* $\neg(\Box(h \to \bigcirc(p)) \wedge \Box(m \to \bigcirc(\neg p))$.

Next, we use an objective function to measure how "close" the formula is to a BC. According to the three properties of BCs, Degiovanni *et al.* [9] proposed an objective function to capture the similarity between formula and BC. Here we follow their objective function. Given an LTL formula φ, the objective function f is defined as:

$$f(\varphi) = \mathrm{li}(\varphi) + \sum_{i=1}^{|G|} \min(\varphi, G_{-i}) + \mathrm{nt}(\varphi) + \frac{1}{|\varphi|}$$

where $|G|$ is the number of the goals and the functions li, min and nt are defined as follows:

$$li(\varphi) = \begin{cases} 1 & \text{if } (Dom \wedge G \wedge \varphi) \text{ is UNSAT} \\ 0 & \text{otherwise} \end{cases}$$

$$min(\varphi, G_{-i}) = \begin{cases} \frac{1}{|G|} & \text{if } (Dom \wedge G_{-i} \wedge \varphi) \text{ is SAT} \\ 0 & \text{otherwise} \end{cases}$$

$$nt(\varphi) = \begin{cases} 0.5 & \text{if } \neg G \not\equiv \varphi \\ 0 & \text{otherwise} \end{cases}$$

Algorithm 1: The LOGION Algorithm

Input: the domain properties Dom and a set of goals $G_1, \ldots, G_n$
Output: a set of boundary conditions $\mathfrak{B}$

1 $\mathfrak{B} \leftarrow \emptyset$;
2 $\varphi* \leftarrow \neg(G_1 \wedge \cdots \wedge G_n)$;
3 **while** *the cutoff time is not reached* **do**
4 **if** $N(\varphi*, T) \neq \emptyset$ **then**
5 $\bar{\mathfrak{B}} \leftarrow N_k(\varphi*, T)$;
6 $\varphi* \leftarrow$ the formula in $\bar{\mathfrak{B}}$ with highest score;
7 $\mathfrak{B} \leftarrow \mathfrak{B} \cup$ BCs in $\bar{\mathfrak{B}}$;
8 T-tabu update;
9 **return** $\mathfrak{B}$;

Intuitively, the first three terms of the objective function f capture the three properties of BC, respectively: *li* for the logical inconsistency, *min* for the minimality and *nt* for the non-triviality. The last term makes the local search tend to produce compact formulae, which is a secondary issue. We use an LTL SAT checker, such as Aalta [21], to check the satisfiability of LTL formula in the objective function.

C. The Description of LOGION

In this section, we give the entire description of the LOGION algorithm based on the components described above. LOGION is shown in Algorithm 1. In the beginning, LOGION uses the trivial condition to construct an initial LTL formula $\varphi*$ (line 2). After that, LOGION executes a search step iteratively until the time budget is reached. During the searching procedure, $\varphi*$ represents the current best formula. $N_k(\varphi*, T)$ is not always empty because T-tabu formulae update at the end of each iteration. LOGION computes the score of each formula in $\bar{\mathfrak{B}}$ according to the objective function mentioned in IV-B, and walks to the formula with the highest score (line 6). LOGION adds all BCs from $\bar{\mathfrak{B}}$ into $\mathfrak{B}$ (line 7). Finally, it returns all BCs $\bar{\mathfrak{B}}$ (line 9).

V. Experiments

In this section, we conduct extensive experiments on a broad range of cases shown in Section III to evaluate the effectiveness of our LOGION algorithm by comparing it with the state-of-the-art approaches. We start by presenting two research questions and presenting the state-of-the-art competitors and experimental preliminaries about the experiments. Then, we show the experimental results and give some discussions about the empirical results. Finally, we show an application of BC.

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

RQ2: To what extent does LOGION exploit the structural similarity of BCs?

RQ3: How does LOGION compare against the state-of-the-art competitors on cases?

A. The State-of-the-art Competitors

We compare LOGION against two state-of-the-art approaches. One [9] is based on a genetic algorithm, the other [8] is a Tableaux-based LTL satisfiability procedure. we call them as the GA-based approach and the Tableaux-based approach, respectively.

The GA-based approach [9] is effective in finding multiple BCs in different forms since it applies suitable crossover and mutation operators.

The Tableaux-based approach [8] is a deterministic algorithm. It constructs paths that "escape" from the tableau structure and generates a small number of BCs.

B. Experimental Settings

LOGION is implemented in Java, which uses the Java Metaheuristics Search Framework (JAMES) [22], [23], and integrating the LTL2Büchi library [24] to parse LTL requirements specifications, and the LTL satisfiability checker Aalta [21]. In our experiments, the tabu tenure T is set to 4, k is set to 50.

We run the GA-based approach [1] and LOGION 10 times on each of the 16 cases. In addition, we run Tableaux-based approach[2] only one time because it is a deterministic algorithm. The cutoff time for our approach LOGION and the GA-based approach is set to 1 hour, while the cutoff time for the Tableaux-based approach is set to 3 hours.

All the experiments were run on the 2.13GHz Intel E7-4830, with 128 GB memory under GNU/Linux (Ubuntu 16.04).

To compare the more general BCs among different algorithms, we first computed a set of BCs Π that fulfills the following three requirements. (1) Π is composed of BC obtained by all algorithms, (2) $\nexists \psi, \varphi \in \Pi \; s.t. \; \psi \rightarrow \varphi$, and (3) for each BC identified by each algorithm, it is either in Π or less general than a BC in Π. Intuitively, Π covers all the circumstances captured by all BCs identified by each algorithm. Then, we counted the number of BCs in Π identified by each algorithm, denoted by "#gen.". The larger "#gen." means that the BCs identified by the algorithm as a whole is more general than the ones identified by other algorithms.

In addition, we use the method in [25] to evaluate the likelihood of the BCs. In practice, the likelihood can be used to prioritize conflicts to be resolved, and suggest which goals to drive attention to for refinements. Specifically, we first computed the likelihood of the more general BCs. Then, based on the likelihood, we ranked BCs: the BC with the larger likelihood ranks higher. We reported the average ranking of the more general BCs with the highest likelihood of each competitor, denoted by "rank".

C. Experimental Results

In this subsection, we present the experimental results and then answer the research questions mentioned above.

Experiments on to what extent LOGION exploits the structural similarity of BCs (RQ2): For each case study, we

[1] http://dc.exa.unrc.edu.ar/staff/rdegiovanni/ASE2018.html
[2] https://dc.exa.unrc.edu.ar/staff/rdegiovanni/ase2016

TABLE III
EXPERIMENTAL RESULTS ON THE SIMILARITY BETWEEN BOUNDARY CONDITIONS SUCCESSIVELY COMPUTED BY LOGION

Case	LOGION	
	$\delta(BC_i, BC_{i+1})$	$\Delta(BC_i, BC_{i+1})$
RP1	6.43	0.18
RP2	5.70	0.21
Ele	4.72	0.25
TCP	5.24	0.23
AAP	5.32	0.31
MP	4.69	0.21
ATM	5.79	0.22
RRCS	5.70	0.19
Tel	5.39	0.24
LAS	5.75	0.09
PA	4.36	0.09
RRA	5.36	0.14
SA	5.70	0.10
LB	10.43	0.05
LC	17.72	0.10
AMBA	24.67	0.09

"$\delta(BC_i, BC_{i+1})$" stands for the average distance between boundary condition BC_i and its next boundary condition BC_{i+1} successively computed. "$\Delta(BC_i, BC_{i+1})$" means the average normalized distance between boundary condition BC_i and its next boundary condition BC_{i+1} successively computed.

compute the average distances ($\delta(BC_i, BC_{i+1})$) and normalized distance ($\Delta(BC_i, BC_{i+1})$) between the average distance between boundary condition BC_i and its next boundary condition BC_{i+1} computed successively by our algorithm LOGION. The results are shown in Table III. From the table, we can observe that in most cases (except LB, LC and AMBA), the BC_i and BC_{i+1} successively computed are quite similar. In other words, it is possible that once a BC is found, a new BC may be found after two to five neighbor jumps. For LB, LC and AMBA cases, the normalized distances are less than 0.1 but it needs more neighbor jumps to find the next BCs because the BCs have a bigger size.

The similarity between the pairs of BCs computed successively reflects that our algorithm LOGION in fact exploits the structural similarity to find BCs.

Experiments on comparing LOGION against its state-of-the-art competitors (RQ3): The results are shown in Table IV present that LOGION performs substantially better on all cases than Tableaux-based approach and GA-based approach. Firstly, LOGION can handle more cases than others in one hour. Secondly, comparing the BCs found ("#BCs"), LOGION outperforms the GA-based approach and the Tableaux-based approach by one order of magnitude in almost all cases. Thirdly, LOGION generally have higher "#gen." and "rank" than others. It indicates that the local search algorithm exhibits a faster search process to search for BCs than the tableaux and genetic algorithm. LOGION potentially identifies more general BCs based on a large among of BCs. This shows that LOGION not only finds more BCs but also more general BCs. Fourthly, for the first 15 cases, we can observe that the success rate of LOGION is higher than that of the two approaches, which demonstrates that our approach LOGION is more robust.

Besides, we reported the time of identifying the first BC ("T_{FBC}") and the size of the best BC (the most compact BC "S_{BBC}"). LOGION not only identifies first BCs fast but also performs better search bias towards compact formulae.

We notice that none of the approaches identified BCs on the AMBA. We think that the reason is as follows. For the Tableaux-based approach, it cannot generate the tableau structure of AMBA within 3 hours. Actually, it can only generate tableau structure of RP1, RP2, TCP, AAP, and ATM within 3 hours. For the GA-based approach and LOGION, to verify whether a formula is BC in AMBA, they need to call the LTL satisfiability checker 24 times (logical inconsistency 1 time, minimality 21 times, and non-trivial 2 times). Each time the size of the formula checked by the LTL satisfiability checker is large. Therefore, both approaches can only search for a small number of formulae within an hour on AMBA, resulting in not finding BCs.

In short, LOGION significantly outperforms the state-of-the-art approaches, especially in computing BCs and general BCs.

D. Application

The direct application of BC is used to repair requirements specifications. One of the common strategies for resolving divergence is goal weakening. Its principle is to weaken the formulation of one or several among the divergent goals so as to make divergence disappear. For Example 1, Emmanuel Letier et al [26] resolved divergence by weakening the first goal to cover BC_2 ($\Diamond(h \wedge m)$). The first goal after the change is "the pump is switched on when the water level is high and there is no methane". Formally, $\Box((h \wedge \neg m) \rightarrow \bigcirc(p))$.

BCs are also used to explain the synthesis unrealizability. For Example 1, we ask some synthesis tools, like Ratsy [27], to build a controller that satisfies the specified goals. We will get as an answer that the specification is unrealizable. Recall that two BCs are presented in example 1. These formulae give us information about some admissible behaviors of the system, that lead us to violate the goals. It means that the environment always has a winning strategy to make the controller reach the BC. Such a BC could be thought of as an explanation of why the controller cannot satisfy all goals.

VI. RELATED WORK

Besides the inconsistency management approaches based on the informal or semi-formal way, such as [28]–[31], a series of formal approaches [4], [32]–[34] recently have been proposed, which only focus on logical inconsistency or ontology mismatch. Another related approach is Nuseibeh and Russo's work [35], which generates a conjunction of ground literals as an explanation for the unsatisfiable specification based on abduction reasoning. For consistency checking methods, we have to mention the approach of Harel *et al.* [33], which identifies inconsistencies between two requirements represented as conditional scenarios. While in this paper, we are interested in identifying the situations that lead to goal divergences, which

TABLE IV
EXPERIMENTAL RESULTS OF LOGION, THE GA-BASED APPROACH AND THE TABLEAUX-BASED APPROACH (TABLEAUX FOR SHORT)

Case	Tableaux				GA-based Approach						LOGION					
	#BC	#gen.	rank	Time	#BC	#gen.	rank	T_{FBC}	S_{BBC}	#suc.	#BC	#gen.	rank	T_{FBC}	S_{BBC}	#suc.
RP1	1.0	0.9	2.9	0.1	45.2	2.8	**1.4**	8.8	14.2	**10**	**6935.5**	**4.0**	1.5	**3.5**	**8.8**	**10**
RP2	1.0	0.7	1.6	0.4	48.0	3.0	1.4	18.7	16.8	**10**	**14158.0**	**7.0**	**1.3**	**1.4**	**7.6**	**10**
Ele	-	-	-	-	41.2	2.1	**1.5**	4.1	**5.0**	**10**	**7759.4**	**22.2**	**1.5**	**1.8**	**5.0**	**10**
TCP	2.0	0.6	2.2	1.0	80.4	0.4	2.0	17.2	**5.1**	**10**	**10335.4**	**20.0**	**1.0**	**0.9**	7.3	**10**
AAP	4.0	4.0	2.3	0.3	56.8	1.1	2.2	25.1	**3.0**	**10**	**22512.9**	**41.9**	**1.1**	**4.4**	**3.0**	**10**
MP	-	-	-	-	75.2	3.0	1.9	10.0	**3.0**	**10**	**27112.2**	**26.9**	**1.1**	**0.3**	**3.0**	**10**
ATM	3.0	2.0	2.4	1.9	98.6	1.4	1.9	12.0	**6.0**	**10**	**8773.6**	**88.3**	**1.4**	**0.6**	12.4	**10**
RRCS	-	-	-	-	60.4	4.2	1.6	7.5	11.0	**10**	**17912.0**	**113.4**	**1.2**	**1.1**	**6.4**	**10**
Tel	-	-	-	-	155.0	4.2	2.0	61.3	12.7	9	**12408.2**	**89.1**	**1.0**	**1.0**	**6.6**	**10**
LAS	-	-	-	-	36.8	0.9	**1.2**	913.3	43.0	6	**12125.4**	**133.3**	1.3	**1.2**	**20.6**	**10**
PA	-	-	-	-	-	-	-	-	-	0	**3220.6**	**45.9**	**1.0**	**2.4**	**15.4**	**10**
RRA	-	-	-	-	88.9	0.4	2.0	470.1	12.6	**10**	**1966.5**	**80.2**	**1.0**	**1.2**	**10.9**	**10**
SA	-	-	-	-	-	-	-	-	-	0	**3600.8**	**42.9**	**1.0**	**11.3**	**20.7**	**10**
LB	-	-	-	-	-	-	-	-	-	0	**87.8**	**2.8**	**1.0**	**1489.2**	**62.5**	**4**
LC	-	-	-	-	-	-	-	-	-	0	**16.6**	**1.7**	**1.0**	**389.3**	**91.6**	**7**
AMBA	-	-	-	-	-	-	-	-	-	0	-	-	-	-	-	0

"#BC" stands for the average number of BCs in 10 runs. The definitions of "#gen." and "rank" are shown in Section V-B. "T_{FBC}" means the time (second) of identifying the first BC. "S_{BBC}" means the size of the best BC (the most compact BC). "#suc." is the number of successful runs (out of 10 runs). Finally, "-" means the failed case.

are nothing but weak inconsistencies. The works on reasoning about conflicts in requirements also include [36]–[38].

For the assessment of conflicts, Degiovanni *et al.* [25] recently have proposed an automated approach to assess how likely conflict is, under an assumption that all events are equally likely. For the resolution of conflicts, Murukannaiah *et al.* [39] resolved the conflicts among stakeholder goals of system-to-be based on the Analysis of Competing Hypotheses technique and argumentation patterns. Related works on conflict resolution also include [40] which calculates the personalized repairs for the conflicts of requirements with the principle of Model-based Diagnosis. However, these approaches presuppose that the conflicts have been already identified and our approach for BC discovery provides a footstone for solving these problems.

In goal-oriented requirements engineering, we have to mention the work on obstacle analysis. An obstacle, first proposed in [41], is a particular goal conflict, which captures the situation that only one goal is inconsistent with the domain properties. Alrajeh *et al.* [42] exploited model checking technique to generate tracks that violate or satisfy the goals, and then compute obstacles from these tracks based on the machine learning technique. Other approaches of obstacle analysis include [41], [43]–[45]. Whereas, as obstacles only capture the inconsistency for single goals, these approaches fail to deal with the situations where multiple goals are conflicting.

Let us come back to the goal-conflict identification problem. The concept of goal conflict was first proposed by van Lamsweerde *et al.* [7], who also proposed a pattern-based approach to identify a goal conflict in a requirement specification. But the syntactical restrictions on the goal specifications and the ability of computing only one BC indeed limit the applicability of the approach. While our approach LOGION has no limitation on the specifications and is able to generate BCs in any form theoretically.

In 2016, Degiovanni *et al.* [8] paid attention on goal-conflict identification again. They provided a tableaux-based approach to generate BCs, consisting of two phases. It first generates a conjunction of domain properties and goals ($Dom \wedge G$) via a tableau, then identifies BCs with a complex logical algorithm based on tableaux. However, for the specifications with a large number of domain properties and goals, the approach suffers from an efficiencies issue because tableaux are difficult to be generated. It limits the approach to be applicable only on small specifications. As shown in the last section, our approach LOGION, as an anytime algorithm, generates significantly more BCs and solves more cases within the same time interval.

Another related work lies in Degiovanni *et al.* [9] which applies a genetic algorithm to BCs. Based on the definition of BCs, they defined a fitness function to guide towards finding compact BCs. However, our approach is more efficient than the GA-based approach and generates significantly more BCs. Because the neighborhood relation of formulae captures the structural similarity, our approach finds another BCs within a few iterations once a BC is found.

VII. CONCLUSION AND FUTURE WORK

In this paper, we discover a frequent phenomenon that some BCs are similar on the formula structure and give a formal analysis *w.r.t.* the formula distance. Based on such an observation, we present an efficient local search algorithm LOGION, to automatically identify BCs, which is featured as capturing the similarity in formula structure of BCs. By taking experiments on the classical cases, we show that our approach LOGION is more efficient than the state-of-the-art approaches of computing BCs and general BCs. In future work, we hope to optimize the BC verification procedure by reducing the calls of the LTL satisfiability checker.

ACKNOWLEDGMENT

We thank Shaowei Cai for the discussion on the paper; Xiaotong Song for her help on the experiments. This paper was supported by the Guangdong Province Science and Technology Plan projects (No. 2017B010110011 and 2016B030305007), National Natural Science Foundation of China (No. 61976232; 61573386; 61906216), National Key R&D Program of China (No. 2018YFC0830600), Guangdong Province Natural Science Foundation (No. 2016A030313292, 2017A070706010 (soft science), and 2018A030313086), Guangdong Basic and Applied Basic Research Foundation (No. 2020A1515010642), Guangzhou Science and Technology Project (No. 201804010435), and the Fundamental Research Funds for the Central Universities (19lgpy226).

REFERENCES

[1] A. Chakraborty, M. K. Baowaly, A. Arefin, and A. N. Bahar, "The role of requirement engineering in software development life cycle," *Journal of emerging trends in computing and information sciences*, vol. 3, no. 5, pp. 723–729, 2012.

[2] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel, "Learning operational requirements from goal models," in *ICSE*, 2009, pp. 265–275.

[3] R. Degiovanni, D. Alrajeh, N. Aguirre, and S. Uchitel, "Automated goal operationalisation based on interpolation and sat solving," in *ICSE*, 2014, pp. 129–139.

[4] C. Ellen, S. Sieverding, and H. Hungar, "Detecting consistencies and inconsistencies of pattern-based functional requirements," in *FMICS*, 2014, pp. 155–169.

[5] A. Van Lamsweerde, *Requirements engineering: From system goals to UML models to software*. Chichester, UK: John Wiley & Sons, 2009, vol. 10.

[6] J. Kramer, J. Magee, M. Sloman, and A. Lister, "Conic: an integrated approach to distributed computer control systems," *IET Computers & Digital Techniques*, vol. 130, no. 1, pp. 1–10, 1983.

[7] A. Van Lamsweerde, R. Darimont, and E. Letier, "Managing conflicts in goal-driven requirements engineering," *IEEE Trans. Software Eng.*, vol. 24, no. 11, pp. 908–926, 1998.

[8] R. Degiovanni, N. Ricci, D. Alrajeh, P. Castro, and N. Aguirre, "Goal-conflict detection based on temporal satisfiability checking," in *ASE*, 2016, pp. 507–518.

[9] R. Degiovanni, F. Molina, G. Regis, and N. Aguirre, "A genetic algorithm for goal-conflict identification," in *ASE*, 2018, pp. 520–531.

[10] A. Van Lamsweerde and E. Letier, "Integrating obstacles in goal-driven requirements engineering," in *ICSE*, 1998, pp. 53–62.

[11] E. A. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science*. Elsevier, 1990, pp. 995–1072.

[12] A. P. Sistla and E. M. Clarke, "The complexity of propositional linear temporal logics," *J. ACM*, vol. 32, no. 3, pp. 733–749, 1985.

[13] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems: Specification*. Springer Science & Business Media, 2012.

[14] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM J. Comput*, vol. 18, no. 6, pp. 1245–1262, 1989.

[15] J. R. Rico-Juan and L. Micó, "Comparison of aesa and laesa search algorithms using string and tree-edit-distances," *Pattern Recognition Letters*, vol. 24, no. 9-10, pp. 1417–1426, 2003.

[16] P. Bille, "A survey on tree edit distance and related problems," *Theor. Comput. Sci.*, vol. 337, no. 1-3, pp. 217–239, 2005.

[17] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computers & OR*, vol. 13, no. 5, pp. 533–549, 1986.

[18] P. Hansen and B. Jaumard, "Algorithms for the maximum satisfiability problem," *ACM Journal of Experimental Algorithmics*, vol. 44, no. 4, pp. 279–303, 1990.

[19] H. H. Hoos and T. Stützle, *Stochastic local search: Foundations and applications*. Elsevier, 2004.

[20] H. Enderton and H. B. Enderton, *A mathematical introduction to logic*. Elsevier, 2001.

[21] J. Li, S. Zhu, G. Pu, and M. Y. Vardi, "Sat-based explicit ltl reasoning," in *HVC*, 2015, pp. 209–224.

[22] H. De Beukelaer, G. F. Davenport, G. De Meyer, and V. Fack, "James: An object-oriented java framework for discrete optimization using local search metaheuristics," *Softw., Pract. Exper.*, vol. 47, no. 6, pp. 921–938, 2017.

[23] ——, "James: A modern object-oriented java framework for discrete optimization using local search metaheuristics," in *Operational Research*, 2015, pp. 134–138.

[24] D. Giannakopoulou and F. Lerda, "From states to transitions: Improving translation of ltl formulae to büchi automata," in *FORTE*, 2002, pp. 308–326.

[25] R. Degiovanni, P. Castro, M. Arroyo, M. Ruiz, N. Aguirre, and M. Frias, "Goal-conflict likelihood assessment based on model counting," in *ICSE*, 2018, pp. 1125–1135.

[26] E. Letier *et al.*, "Reasoning about agents in goal-oriented requirements engineering," Ph.D. dissertation, PhD thesis, Université catholique de Louvain, 2001.

[27] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, and R. Seeber, "Ratsy–a new requirements analysis tool with synthesis," in *International Conference on Computer Aided Verification*, 2010, pp. 425–429.

[28] J. H. Hausmann, R. Heckel, and G. Taentzer, "Detection of conflicting functional requirements in a use case-driven approach," in *ICSE*, 2002, pp. 105–115.

[29] S. J. Herzig and C. J. Paredis, "A conceptual basis for inconsistency management in model-based systems engineering," *Procedia CIRP*, vol. 21, pp. 52–57, 2014.

[30] M. Kamalrudin, "Automated software tool support for checking the inconsistency of requirements," in *ASE*, 2009, pp. 693–697.

[31] M. Kamalrudin, J. Hosking, and J. Grundy, "Improving requirements quality using essential use case interaction patterns," in *ICSE*, 2011, pp. 531–540.

[32] N. A. Ernst, A. Borgida, J. Mylopoulos, and I. J. Jureta, "Agile requirements evolution via paraconsistent reasoning," in *CAiSE*, 2012, pp. 382–397.

[33] D. Harel, H. Kugler, and A. Pnueli, "Synthesis revisited: Generating statechart models from scenario-based requirements," in *Formal Methods in Software and Systems Modeling*. Springer, 2005, pp. 309–324.

[34] T. H. Nguyen, B. Q. Vo, M. Lumpe, and J. Grundy, "Kbre: a framework for knowledge-based requirements engineering," *Software Quality Journal*, vol. 22, no. 1, pp. 87–119, 2014.

[35] B. Nuseibeh and A. Russo, "Using abduction to evolve inconsistent requirements specification," *Australasian J. of Inf. Systems*, vol. 7, no. 1; SPI, pp. 118–130, 1999.

[36] I. J. Jureta, A. Borgida, N. A. Ernst, and J. Mylopoulos, "Techne: Towards a new generation of requirements modeling languages with goals, preferences, and inconsistency handling," in *RE*, 2010, pp. 115–124.

[37] C.-L. Liu, "Ontology-based conflict analysis method in non-functional requirements," in *ACIS-ICIS*, 2010, pp. 491–496.

[38] D. Mairiza and D. Zowghi, "Constructing a catalogue of conflicts among non-functional requirements," in *ENASE*, 2010, pp. 31–44.

[39] P. K. Murukannaiah, A. K. Kalia, P. R. Telangy, and M. P. Singh, "Resolving goal conflicts via argumentation-based analysis of competing hypotheses," in *RE*, 2015, pp. 156–165.

[40] A. Felfernig, G. Friedrich, M. Schubert, M. Mandl, M. Mairitsch, and E. Teppan, "Plausible repairs for inconsistent requirements," in *IJCAI*, 2009, pp. 791–796.

[41] A. Van Lamsweerde and E. Letier, "Handling obstacles in goal-oriented requirements engineering," *IEEE Trans. Software Eng.*, vol. 26, no. 10, pp. 978–1005, 2000.

[42] D. Alrajeh, J. Kramer, A. Van Lamsweerde, A. Russo, and S. Uchitel, "Generating obstacle conditions for requirements completeness," in *ICSE*, 2012, pp. 705–715.

[43] A. Cailliau and A. Van Lamsweerde, "A probabilistic framework for goal-oriented risk analysis," in *RE*, 2012, pp. 201–210.

[44] A. Cailliau and A. van Lamsweerde, "Integrating exception handling in goal models," in *RE*, 2014, pp. 43–52.

[45] ——, "Handling knowledge uncertainty in risk-based requirements engineering," in *RE*, 2015, pp. 106–115.