A Genetic Algorithm for Goal-Conflict Identification

Renzo Degiovanni
Universidad Nacional de Río Cuarto
Argentina
rdegiovanni@dc.exa.unrc.edu.ar

Facundo Molina
Universidad Nacional de Río Cuarto
and CONICET, Argentina
fmolina@dc.exa.unrc.edu.ar

Germán Regis
Universidad Nacional de Río Cuarto
Argentina
gregis@dc.exa.unrc.edu.ar

Nazareno Aguirre
Universidad Nacional de Río Cuarto
and CONICET, Argentina
naguirre@dc.exa.unrc.edu.ar

ABSTRACT

Goal-conflict analysis has been widely used as an abstraction for risk analysis in goal-oriented requirements engineering approaches. In this context, where the expected behaviour of the system-to-be is captured in terms of domain properties and goals, identifying combinations of circumstances that may make the goals diverge, i.e., not to be satisfied as a whole, is of most importance.

Various approaches have been proposed in order to automatically identify *boundary conditions*, i.e., formulas capturing goal-divergent situations, but they either apply only to some specific goal expressions, or are affected by scalability issues that make them applicable only to relatively small specifications. In this paper, we present a novel approach to automatically identify boundary conditions, using evolutionary computation. More precisely, we develop a genetic algorithm that, given the LTL formulation of the domain properties and the goals, it searches for formulas that capture divergences in the specification. We exploit a modern LTL satisfiability checker to successfully guide our genetic algorithm to the solutions. We assess our technique on a set of case studies, and show that our genetic algorithm is able to find boundary conditions that cannot be generated by related approaches, and is able to efficiently scale to LTL specifications that other approaches are unable to deal with.

CCS CONCEPTS

• **Software and its engineering** → **Requirements analysis**; *Risk management*; *Search-based software engineering*; • **Theory of computation** → *Modal and temporal logics*;

KEYWORDS

Goal Conflicts, Genetic Algorithms, LTL Satisfiability

ACM Reference Format:
Renzo Degiovanni, Facundo Molina, Germán Regis, and Nazareno Aguirre. 2018. A Genetic Algorithm for Goal-Conflict Identification. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238220

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238220

1 INTRODUCTION

The requirements process is a key stage in many software development processes, whose main concerns are the correct understanding of the problem to be solved by the system-to-be (as well as the corresponding problem domain), and its detailed specification [20, 27, 52]. In general, understanding the objectives of a system to be developed is not straightforward, and demands comprehensive elicitation activities, to arrive to an explicit requirements specification. And arriving to a requirements specification is an important milestone in the requirements process, that enables a number of further analysis tasks that can be performed on the specification, to understand the interactions between different system goals, potential contradictions and obstacles to requirements fulfilment, etc. [52].

When requirements are expressed using some *formal* language, specifications can often be subject to certain automated analyses, to find flaws and imprecisions. For instance, checking for requirements satisfiability corresponds to ascertaining the absence of contradictions in requirements. But such an analysis is only able to find the simplest kinds of problems in requirements satisfaction; many times requirements are indeed satisfiable as a whole, but admit situations where goals *diverge*, i.e., where the satisfaction of some system goals inhibits the satisfaction of others. Identifying these circumstances early in the development process is of most importance, since it enables one to improve specifications, take countermeasures to these situations, and more deeply understand the roots for potential system goal unsatisfiability.

Various approaches have been proposed in order to automatically identify goal-divergent situations in the context of goal-oriented requirements engineering. These divergences, known as *boundary conditions*, have some particularities, that make them difficult to detect. Firstly, they capture goal divergences through *formulas*, not simple specification states, that are consistent with the domain properties, but imply the violation of the system goals. Secondly, they characterise *subtle* goal violations, in the sense that they cannot be simple negations of goals, and they only lead to the unfulfilment of the goals when these are jointly considered, i.e., if a single goal is removed, the whole situation becomes satisfiable. Existing techniques to identify boundary conditions have limitations. Some techniques follow a pattern-based mechanism

for boundary condition identification [53], and thus only apply to some specific goal expressions, when these are specified in a certain syntactic way. Other approaches are based on more sophisticated logical mechanisms, that process certain semantic constructions generated from goals and domain properties, in order to identify boundary conditions [13]; these, on the other hand, are affected by scalability limitations of the underlying logical mechanisms, that make them applicable only to relatively small specifications.

In this paper, we present a novel approach to automatically identify boundary conditions, using evolutionary computation. This approach outperforms related techniques through the development of a genetic algorithm that, given the LTL formulation of the domain properties and the goals, searches for formulas that capture divergences in the specification. This algorithm exploits a modern LTL satisfiability checker to successfully guide the genetic search toward suitable solutions. Our evaluation, based on various case studies taken from the literature, shows that this evolutionary mechanism is able to find boundary conditions that cannot be generated by related approaches, and is able to efficiently scale to LTL specifications that other approaches are unable to deal with.

The remainder of the paper is organised as follow. Section 2 introduces preliminary concepts about Goal-Oriented Requirements, Linear-Time Temporal Logic and Genetic Algorithms, necessary in the paper. Section 3 presents an illustrating example, used to motivate the approach. Section 4 describes the approach in detail. In Section 5 we validate our technique, by applying it to various case studies, as well as comparing the approach with related techniques in terms of efficiency and effectiveness. Finally, we discuss related work in Section 6, and draw some conclusions and describe lines of further work in Section 7.

2 BACKGROUND

2.1 Goal-Oriented Requirements

Goal-Oriented Requirements Engineering (GORE) [52] drives the requirements process in software development from the definition of high-level goals, that state how the system to be developed should behave. Goals are prescriptive statements that the system must achieve through the collaboration of cooperating agents, that might include humans, hardware devices and of course the software system, within a given domain. This domain must also be explicitly characterised, via *descriptive* statements about the problem world, such as natural laws, collectively referred to as *domain properties*. Within this context, a goal model consists of a decomposition of goals via refinements, capturing how a goal can be fulfilled in terms of simpler ones. Goal refinement terminates when every leaf subgoal can be assigned to a single agent, that will be in charge of guaranteeing its achievement (agents feature operations, through which they must fulfil the goals).

Generally, the description of software requirements can be inadequate for various reasons. For instance, assuming an unrealistic benevolent behaviour of the environment can make the goals too ideal to be met. Also, some unanticipated cases can make a requirements specification incomplete. Even the goals themselves may be inconsistent as a whole, i.e., they may not be jointly satisfiable.

In GORE methodologies, dealing with the above-cited kind of problems, as early as possible, is of most importance. The *conflict analysis phase* [52, 54] deals with these issues, through three main stages: *(1)* the *identification* stage, which consists of identifying conflicts between goals (i.e., conditions that, when present, make the goals inconsistent); *(2)* the *assessment* stage, consisting of assessing and prioritising the identified conflicts according to their likelihood and severity; and *(3)*, the *resolution* stage, where conflicts are resolved by providing appropriate countermeasures and, consequently, transforming the goal model, guided by the criticality level obtained during assessment.

This paper focuses on the *identification* stage, with the provision of an automated mechanism for goal conflict discovery. A *conflict* essentially represents a condition whose occurrence results in the loss of satisfaction of the goals, i.e., that makes the goals *diverge* [53]. More formally, given a set $G_1, \ldots, G_n$ of goals and a set *Dom* of domain properties, these are said to be *divergent* if and only if there exists an expression *BC*, called a *boundary condition*, such that the following conditions hold:

$$\{Dom, BC, \bigwedge_{1 \le i \le n} G_i\} \models false, \qquad (\textit{logical inconsistency})$$

$$\{Dom, BC, \bigwedge_{j \ne i} G_j\} \not\models false, \text{for each } 1 \le i \le n \qquad (\textit{minimality})$$

$$BC \neq \neg(G_1 \wedge \ldots \wedge G_n) \qquad (\textit{non-triviality})$$

Intuitively, a boundary condition captures a particular combination of circumstances in which the goals cannot be satisfied as a whole. The first condition establishes that, when *BC* holds, the conjunction of goals $G_1, \ldots, G_n$ becomes inconsistent. The second condition states that, if any of the goals is disregarded, then consistency is recovered. The third condition prohibits a boundary condition to be simply the negation of the goals. Also, due to the minimality condition, it cannot be *false* (it has to be consistent with the domain *Dom*). Section 3 provides an illustrating example, that further explains the intuition behind boundary conditions.

Typically, formal requirements engineering methodologies adopt a logical formalism to precisely capture the desired system goals and domain properties. For instance, the KAOS method [52] uses Linear-Time Temporal Logic [38] for formally specifying software requirements. Employing a formal language to specify software requirements enables the use of (semi-)automated analysis mechanisms, to assess specifications. For instance, if LTL formulas are used to specify requirements, one may use automated LTL satisfiability solvers to check for the feasibility of the corresponding requirements. As we will describe in detail later on, we will exploit efficient LTL satisfiability solvers to automatically check whether generated candidate formulas satisfy or not the conditions to be valid boundary conditions.

2.2 Linear-Time Temporal Logic

Linear-Time Temporal Logic (LTL) [38] is a logical formalism that has been extensively employed to state properties of reactive systems, and more recently, for specifying software requirements [52]. LTL assumes that the structure of time is linear, i.e., each instant of time is followed by a unique future instant. The syntax of LTL formulas is inductively defined using a set *AP* of propositional variables, the standard logical connectives and temporal operators $\bigcirc$ and $\mathcal{U}$, as follows: *(i)* $b \in \mathbb{B}$ is an LTL formula, where $\mathbb{B} = \{true, false\}$; *(ii)* every proposition $p \in AP$ is an LTL formula, and *(iii)* if φ_1 and φ_2

are LTL formulas, then so are $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2$, $\bigcirc\varphi_1$ and $\varphi_1 \mathcal{U} \varphi_2$. We consider the usual definition for the operators $\Box$ (always), $\Diamond$ (eventually), $\mathcal{R}$ (release) and $\mathcal{W}$ (weak-until) in terms of $\bigcirc$, $\mathcal{U}$, and logical connectives.

LTL formulas are interpreted over infinite traces of propositional valuations. Let σ be an infinite trace. Formulas with no temporal operators are evaluated in the first state of σ. On the other hand, $\bigcirc\varphi$ is true in σ if and only if φ is true in $\sigma[1..]$ (the trace obtained by removing the first state from σ), and $\varphi_1 \mathcal{U} \varphi_2$ is true in σ if and only if there exists a position i such that φ_2 holds in $\sigma[i..]$, and for all $0 \leq j < i$, φ_1 holds in $\sigma[j..]$.

The satisfiability problem for LTL consists of checking, given an LTL formula φ, if there exists at least one trace σ that makes φ hold, i.e., φ evaluates to true in σ. LTL satisfiability is a *decidable* problem [46], and there exist various tools that implement LTL satisfiability checking, so called LTL SAT solvers. As we explain later on in the paper, such tools are central to our evolutionary computation approach, since LTL SAT solving is employed as part of the fitness computation in the search for boundary conditions.

We refer the reader to [37] for further details on linear-time temporal logic.

2.3 Genetic Algorithms

Genetic algorithms [23] are heuristic search algorithms, inspired in natural evolution. As opposed to more traditional search algorithms, that maintain a single "current" candidate solution during the search space traversal, a genetic algorithm operates on a *population* of candidate solutions to a given problem. These candidates are called *individuals* or *chromosomes*, and are often represented as sequences of *genes* (characteristics) that capture their features. A genetic algorithm starts with an initial population of individuals, whose individuals are produced in some arbitrary way, e.g., randomly, and explores the search space by iteratively *evolving* the population, trying to generate a population containing an individual that represents a solution to the problem. At each iteration of this evolution process, members of the current population are selected to make the population evolve, by producing further individuals using two genetic operators: *crossover*, that produces new individuals by combining parts of existing ones, and *mutation*, that creates new individuals by randomly producing changes on existing ones. The selection of individuals to which the genetic operators will be applied, as well as the selection of individuals to be discarded after each iteration, are guided by a *fitness function*. A fitness function is a heuristic function that measures how "fit" a particular individual is, i.e., how close a given candidate is to being an actual solution to the problem being solved. The evolution process is usually performed a defined number of iterations (known as *generations* of the population), or until some termination criterion is met.

As it will be described in later sections, we will employ genetic algorithms to search for boundary conditions of goal-oriented requirements specifications. Thus, individuals will in our case represent LTL formulas, the genetic operators will produce formulas from other formulas, and the fitness function should attempt to evaluate how "close" a formula is to being a boundary condition.

For further details on genetic algorithms, we refer the reader to [40].

3 MOTIVATION

In this section, we will illustrate through a running example both the problem we tackle in this paper, and the main ideas behind our approach based on a genetic algorithm. The example we will use is a simple rail road crossing system (RRCS) [6]. In this model, a train may approach and enter a crossing; these events are captured by two propositions, ta and tc, respectively. A car may also approach and enter the crossing, and these events are captured by ca and cc, respectively. The crossing gate may be opened (go) or closed ($\neg go$). Whenever the train is approaching, the gate should be closed, and it will be reopened after the train has left the crossing. On the other hand, whenever a car approaches the crossing, it will be able to cross only if the gate is open. Let us assume that from the analysis of the above statements, the following goals and domain properties have been elicited:

Domain property: *TrainsDoNotStop*
FormalDef: $\Box(\bigcirc(tc) \leftrightarrow ta)$

Domain property: *CarsCrossWhenGateIsOpened*
FormalDef: $\Box(\bigcirc(cc) \rightarrow ca \wedge go)$

Goal: Avoid[*Collision*]
FormalDef: $\Box\neg(tc \wedge cc)$

Goal: Maintain[*ClosedGateWhenTrainApproaching*]
FormalDef: $\Box(ta \rightarrow \neg go)$

Notice that the specification is *consistent*, i.e., all goals can simultaneously be satisfied, for instance when no train and no car approach the crossing. However, this specification can exhibit some *conflicts*, in particular when the controller opens the gate at the same time that the train is crossing (i.e., $go \wedge tc$), enabling an approaching car to cross as well, and consequently to collision with the train (i.e., $cc \wedge tc$). This conflicting situation can be characterised by a *boundary condition*, in this case $(go \wedge tc)\ \mathcal{W}\ (cc \wedge tc)$. Boundary conditions generalise conflicting situations, by capturing similar contending scenarios by formulas in the same language used to express goals and domain properties. Both identifying conflicting situations and devising corresponding boundary conditions are non-trivial tasks (recall in particular the conditions that formulas must satisfy to be boundary conditions). In this work, we propose using a genetic algorithm to automatically discover boundary conditions from a formal goal model, with goals and domain properties expressed in LTL. The algorithm deals with a very large search space of LTL formulas, built through a syntactic manipulation of the formal domain properties and goals. Let us provide some intuition on how our approach works.

Intuitively, a chromosome in our genetic algorithm represents an LTL formula φ, in such a way that each gene of the chromosome characterises a sub-formula of φ. We consider as the initial population of our genetic algorithm the set of all sub-formulas that can be built from the domain properties and the goals, and their corresponding negations. For instance, from the goal *ClosedGateWhenTrainApproaching* of our running example, the chromosomes that characterise the following 5 sub-formulas are created: $\Box(ta \rightarrow \neg go)$, $ta \rightarrow \neg go$, ta, $\neg go$ and go; as well as their corresponding negations: $\neg\Box(ta \rightarrow \neg go)$, $\neg(ta \rightarrow \neg go)$ and $\neg ta$ (notice that, go and $\neg go$ have already been considered).

In order to obtain new individuals to make the population evolve, some chromosomes are selected at each iteration, and some genetic operators are applied to these. In particular, our genetic algorithm implements the two most common genetic operators, namely, *crossover* and *mutation* operators. Given two chromosomes c_1 and c_2, the crossover operator will produce a new chromosome c_3 using parts of c_1 and c_2. For instance, if chromosomes c_1 and c_2 characterise the LTL formulas *go* and *tc*, respectively, our crossover operator can produce a new chromosome by combining both formulas using some binary operator, such that $go \wedge tc$. On the other hand, given a chromosome c_1, the mutation operator will create a new chromosome c_2 by randomly changing some genes of c_1. For instance, if chromosome c_1 characterises the LTL formula $\Box\neg(tc \wedge cc)$, a particular mutation can be performed, that removes the $\Box$ operator, obtaining the formula $\neg(tc \wedge cc)$.

In a genetic algorithm, the population iteratively evolves guided by a *fitness function*, whose aim is to evaluate individuals, giving higher scores to "better" individuals, i.e., those closer to sought for solutions. This has the aim of guiding the genetic algorithm to generating an individual that represents a solution to the problem being solved. In our case, the fitness function performs a number of SAT calls, to an LTL SAT solver, in order to check, given a chromosome, whether it meets all the conditions to be a boundary condition for the requirements specification or not. At the end of each iteration, those chromosomes with best fitness are selected to move to the next iteration.

Let us consider a specific example, to show how our genetic operators may lead to boundary conditions. Consider the boundary condition $(go \wedge tc)\ \mathcal{W}\ (cc \wedge tc)$ for the specification of the railroad crossing system. The following trace of the genetic algorithm exemplifies how this particular formula may be generated:

(1) Initialize the population with the set of all sub-formulas of the specification, and their negations. In particular, propositions *go* and *tc* will be characterised by some chromosomes c_1 and c_2, respectively.
(2) Then, assume that our algorithm selects both chromosomes c_1 and c_2 to apply the crossover operator, producing a new chromosome c_3; in this particular combination, the $\wedge$ operator is employed, obtaining the LTL formula $(go \wedge tc)$.
(3) Now assume that chromosome c_1' characterising the goal $\Box\neg(tc \wedge cc)$ (it will certainly be in the initial population), is selected for the application of a mutation, in particular one that acts by removing the temporal operator $\Box$; we then obtain a new chromosome c_2' that characterises formula $\neg(tc \wedge cc)$.
(4) Now the algorithm selects chromosome c_2' and applies a mutation operator similar to the previously mentioned, but this time it removes the $\neg$ logical operator, leading to chromosome c_3', that characterises the formula $(tc \wedge cc)$.
(5) Finally, the algorithm selects chromosomes c_3 and c_3' to apply the crossover operator that combines them with the $\mathcal{W}$ operator, obtaining a new chromosome c_4 representing our target boundary condition: $(go \wedge tc)\ \mathcal{W}\ (cc \wedge tc)$.

Of course, the above "trace" is one very specific, of the many that the genetic algorithm may involve. The same boundary condition may be produced with other different paths, and more importantly, many more paths will never produce the boundary condition. To guide it to our desired formulas, the fitness function plays a very important role (as well as the crossover and mutation operators in the definition of the search space), as well as other parameters of the algorithm, like the mutation and crossover rates, population size, individual selection approach, number of generations to consider, etc. The following section will provide further details on all these aspects of our genetic algorithm for boundary condition discovery.

4 A GENETIC ALGORITHM FOR IDENTIFYING BOUNDARY CONDITIONS

As we mentioned in previous sections, the objective of our genetic algorithm is to find situations that capture divergences in the LTL formulation of the requirements specification. In order to express such situations in the same formalism, the search space of our algorithm is composed of LTL formulas. The next sub-subsections detail the representation of the candidate LTL formulas as sequences of genes as well as the main components of the algorithm.

4.1 Chromosome Representation

To represent an LTL formula as a chromosome, i.e., a vector of genes, we first translate the formula to its definitional conjunctive normal form (dCNF) [47]. This gives us a simple and intuitive way of splitting LTL formulas into sub-formulas, in order to generate their corresponding chromosomes.

Given an LTL formula φ, where AP is the set of propositional variables used in φ, let $X = \{x_0, x_1, \ldots\}$ be a set of fresh propositional variables, such that, $AP \cap X = \emptyset$. Then, according to [47], $dCNF_{aux}(\varphi)$ is a set of conjuncts, defined over $AP \cup X$, such that each conjunct represents of a sub-formula ψ of φ. $dCNF_{aux}(\varphi)$ is defined inductively on the structure of the sub-formula ψ as follows:

ψ	Conjunct in $dCNF_{aux}(\varphi)$
b with $b \in \mathbb{B}$	$x_\psi \leftrightarrow b$
p with $p \in AP$	$x_\psi \leftrightarrow p$
$o_1\psi'$ with $o_1 \in \{\neg, \bigcirc, \Diamond, \Box\}$	$x_\psi \leftrightarrow o_1 x_{\psi'}$
$\psi'\ o_2\ \psi''$ with $o_2 \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$	$x_\psi \leftrightarrow x_{\psi'}\ o_2\ x_{\psi''}$

Thus, the *definitional conjunctive normal form* of φ can be defined in the following way:

$$dCNF(\varphi) \equiv x_\varphi \wedge \Box \bigwedge_{c \in dCNF_{aux}(\varphi)} c$$

Given the dCNF representation of φ, we can directly build the chromosome representing φ as follows:

$$[c_1, c_2, ..., c_k] \text{ where each } c_i \in dCNF_{aux}(\varphi)$$

Notice that, as opposed to what is common in genetic algorithms, these chromosomes have varying lengths, since different LTL formulas can have a different amount of conjuncts in their corresponding dCNF representations.

4.2 Initial Population

Since the initial population is a sample of the search space, the routine used for generating it plays an important role. Taking advantage of the formulas present in the domain properties as well as the goals of the given specification, we define the set of formulas $S = Dom \cup G$ and calculate the set of sub-formulas $SF(\psi)$, for each $\psi \in S$, using the following recursive definition:

$\psi = b$ or p with $b \in \mathbb{B}, p \in AP$: $SF(\psi) = \{\psi\}$
$\psi = o_1\psi'$ with $o_1 \in \{\neg, \bigcirc, \Diamond, \Box\}$: $SF(\psi) = \{\psi\} \cup SF(\psi')$
$\psi = \psi' o_2 \psi''$ with $o_2 \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$: $SF(\psi) = \{\psi\} \cup SF(\psi') \cup SF(\psi'')$

Having the set $SF(S)$ of all sub-formulas of each $\psi \in S$, the initial population of individuals is defined to be the following set IP:

$$IP = SF(S) \cup \{\neg s | s \in SF(S)\}$$

Basically, the initial population is the set of all the sub-formulas, as well as their negations, that can be obtained from the domain properties and the goals of the specification.

4.3 Fitness Function

Since each chromosome in the population represents an LTL formula that is a candidate boundary condition for the current specification, the purpose of our fitness function is to evaluate how close is the formula to being an actual boundary condition. By using the definition in Section 2.1, we can exactly determine when a formula is a boundary condition or not. Given a chromosome c of length l_c representing the LTL formula φ_c, the fitness value for c is computed by the following function f:

$$f(c) = li(\varphi_c) + \sum_{i=1}^{|G|} min(\varphi_c, G_i) + nt(\varphi_c) + \frac{1}{l_c}$$

where the functions li, min and nt are defined as follows:

$$li(\varphi_c) = \begin{cases} 1 & \text{if } \{Dom, \varphi_c, \bigwedge_{1 \le i \le n} G_i\} \models false \\ 0 & \text{otherwise} \end{cases}$$

$$min(\varphi_c, G_i) = \begin{cases} \frac{1}{|G|} & \text{if } \{Dom, \varphi_c, \bigwedge_{j \neq i} G_j\} \not\models false \\ 0 & \text{otherwise} \end{cases}$$

$$nt(\varphi_c) = \begin{cases} 0.5 & \text{if } \varphi_c \neq \neg(G_1 \wedge \ldots \wedge G_n) \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, the first three terms of the function f capture the properties that a formula must satisfy in order to be a boundary condition, namely the logical inconsistency (li), the minimality (min) and the non-triviality (nt). With the aim of improving the readability of the produced boundary conditions, the last term of the function applies a penalty related to the formula length, that makes the genetic algorithm to tend to produce smaller formulas. Of course, this is a secondary issue, and this is why it only contributes a fraction to the fitness value, as opposed to the actual driving acceptance criterion, namely, the closeness of the formula to the satisfaction of the properties to be a valid boundary condition.

4.4 Genetic Operators

In order to explore the search space, our genetic algorithm implements a *crossover operator* and a *mutation operator*, both adapted to our chosen chromosome representation.

Given two randomly selected chromosomes c_1 and c_2, representing the LTL formulas φ_1 and φ_2, respectively, our *crossover operator* creates a new chromosome c_n, whose corresponding LTL formula φ_n is calculated by applying one of the following operations:

(1) $\varphi_n = \varphi_1 o_2 \varphi_2$, where $o_2 \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$;
(2) $\varphi_n = \varphi_1[s_2/s_1]$, where $s_1 \in SF(\varphi_1)$ and $s_2 \in SF(\varphi_2)$.

Basically, case (1) corresponds to randomly taking a binary operator $o_2 \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}\mathcal{W}\}$, to create a new LTL formula $\varphi_1 o_2 \varphi_2$, in terms of formulas φ_1 and φ_2. On the other hand, case (2) corresponds to randomly taking a sub-formula s_1 of φ_1 and a sub-formula s_2 of φ_2, and creating a new formula φ_n, based on φ_1, but replacing its sub-formula s_1 by s_2 (i.e., $\varphi_1[s_2/s_1]$). Once the new LTL formula φ_n has been built, the new chromosome c_n is created, as described in Section 4.1.

In contrast to the crossover operator, that applies at the chromosome level, the *mutation operation* applies to randomly selected genes of a chromosome, i.e., it works at the gene level. Recall that each gene of a chromosome characterises a formula φ, that is a particular conjunct of the dCNF representation of φ. Then, genes have the generic form $x_\psi \leftrightarrow \psi$. In order to apply a mutation and maintain a valid dCNF representation, our mutation operation only alters the formula ψ of the gene, obtaining a new formula ψ'. Let g be a gene representing the conjunct $x_\psi \leftrightarrow \psi$. Our mutation operation is defined inductively on the shape of ψ, as follows:

- if $\psi = b$ or $\psi = p$, where $b \in \mathbb{B}$ and $p \in AP$, then:
  (1) $\psi' = b'$, where $b' \in \mathbb{B}$
  (2) $\psi' = r$, where $r \in AP$ and $r \neq p$
  (3) $\psi' = \neg\psi$
- if $\psi = o_1 x_{\psi_1}$, where $o_1 \in \{\neg, \bigcirc, \Diamond, \Box\}$, then:
  (1) $\psi' = x_{\psi_1}$
  (2) $\psi' = o_1' x_{\psi_1}$ where $o_1' \in \{\neg, \bigcirc, \Diamond, \Box\}$ and $o_1' \neq o_1$
  (3) $\psi' = \neg\psi$
- if $\psi = x_{\psi_1} o_2 x_{\psi_2}$, where $o_2 \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$, then:
  (1) $\psi' = x_{\psi_r}$ where $x_{\psi_r} \in \{x_{\psi_1}, x_{\psi_2}\}$
  (2) $\psi' = x_{\psi_1} o_2' x_{\psi_2}$ where $o_2' \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$ and $o_2' \neq o_2$
  (3) $\psi' = \neg\psi$

It is important to note that, for each case of ψ, all the mutations have the same probability to be chosen. In the case that ψ is equal to a boolean value or a proposition, the possible mutations to be applied consist of replacing ψ by *true*, *false*, or by its negation, or by a different proposition. In case that ψ is a unary formula, then the possible mutations to be applied are the deletion of the unary operator from the formula, the replacement of the unary operator by a different one, or just the negation of ψ. Similarly, if ψ is a binary formula, the possible mutations to be applied consist of removing the binary operator, and in this case, one of the operands should be chosen as the new ψ', or replacing the binary operator by a different one, or just negating ψ.

Typically, how many crossovers are applied per generation, and with which probability a gene is mutated, are parameters that one is allowed to configure in a genetic algorithm, to improve effectiveness and efficiency. Our genetic algorithm considers the %10 of the population size to be the number of times that the crossover operator is applied per generation. For instance, if the predefined population size is 100, then our genetic algorithm will perform 10 crossovers per generation. On the other hand, given a chromosome c, with l_c being the number of genes of c, our genetic algorithm considers $1/l_c$ as the probability with which a gene will be mutated. This means that, in the long term, one gene per chromosome will be mutated.

Finally, after applying the fitness function on each chromosome in the population, some of them should be selected to survive to

the next generation. Our algorithm uses a very simple selection operator, based on sorting the individuals of the current population by its fitness values in decreasing order, and selecting as many individuals as the set maximum population size. We also experimented with the use of other known selection algorithms, such as the tournament selector, but in our case studies we always got better results with the "best fitness" selector. In the following section, we assess the performance of our algorithm in case studies of varying complexities, as well as its effectiveness in relation to other techniques.

4.5 Correctness and (In)completeness

Let us now discuss the correctness and (in)completeness of our approach. Regarding correctness, our approach results to be *correct*, i.e., if the genetic algorithm finds a formula that is solution of the problem, then this is indeed a valid boundary condition. Notice that, as it was explained in Section 4.3, the fitness function of the genetic algorithm performs, for each candidate solution, a number of SAT checks in order to determine if the candidate formula satisfies or not all the properties to be a boundary condition. Then, by relying on the correctness of the satisfiability solver for LTL used for this task, Aalta [34] in our case, our genetic algorithm is correct.

Regarding completeness, since our genetic algorithm implements a *non-exhaustive search*, our approach results to be *incomplete*, i.e., there may exist some boundary conditions that are not be visited/considered by our genetic algorithm. However, it is important to remark that our genetic operators are complete, in the sense that, given two formulas φ and ψ over the same set AP of propositions, ψ can be produced from φ by the application of crossover and mutation, provided a sufficiently large chromosome size. Thus, all formulas over the same vocabulary, that fit into the predefined chromosome size, can theoretically be produced by our genetic algorithm.

5 VALIDATION

In this section we evaluate our genetic algorithm with the aim of answering the following research questions:

- RQ1 *How effective and efficient is our approach to identify boundary conditions in requirement specifications?*
- RQ2 *Is our approach able to identify boundary conditions that cannot be derived by related techniques?*

In order to answer RQ1, we consider various requirement specifications taken from the literature and different benchmarks, that feature both safety and liveness goals, and evaluate our genetic algorithm for identifying boundary conditions. In Section 5.1 we present the experimental evaluation on several case studies taken from [5, 13, 53], previously used for assessing a related technique for computing boundary conditions based on a tableaux satisfiability checking algorithm. In addition, we take several case studies used in the Reactive Synthesis Competition (SYNTCOMP) [2], publicly available at [3], which are considerable larger specifications than those taken from the literature that we mentioned before. These larger specifications will be used for assessing the scalability of our genetic algorithm.

To answer RQ2, in Section 5.2 we briefly introduce two previously developed techniques for identifying boundary conditions, namely, a pattern-based approach [53] and a tableaux-based approach [13], and compare the boundary conditions computed by our genetic algorithm against those obtained by the previous techniques.

We analyse the obtained results in Sections 5.3 and 5.4, and discuss the scalability and applicability of our genetic algorithm. In particular, we argue about the different options that one can configure to run the genetic algorithm and how they may affect its efficiency and effectiveness. We also discuss some contexts in which the computed boundary conditions can be used, in addition to the most common one, during the identify-assess-control cycle in the risk analysis of requirements specifications. In particular, we study the possibility of using boundary conditions for explaining the cause of unrealisable specifications, in the context of automated synthesis.

To perform the experimental evaluation, we implemented our genetic algorithm using the Java Genetic Algorithms Package Library (JGAP) [1], and integrating the LTL2Büchi library [21] to parse LTL requirements specifications, and the LTL satisfiability checker Aalta [34], to perform all the SAT checks required by the fitness function. The tool, the specifications for all case studies, and a description of how to reproduce the experiments can be found in the replication package[1]. All the experiments were run on an Intel Core i7 3.2Ghz, with 16Gb of RAM, running GNU/Linux (Ubuntu 16.04).

5.1 Case Studies

We evaluate our genetic algorithm on the following case studies: the Rail Road Crossing System [6], the Mine Pump Controller [32], an elevator controller [15], the ATM [51], the TCP network protocol, the London Ambulance Service (LAS) [19], the Telephone [18], and the three patterns for deriving boundary conditions presented in [53] (Achieve, Retraction1, and Retraction2). Moreover, we consider the specification of a lift controller taken from [5], and some specifications of the SYNTCOMP Repository [3], namely, three variants of the arbiter synchronization protocol (simple, prioritized and round-robin), the ARM's Advanced Microcontroller Bus Architecture (AMBA), and a load balancer protocol for mutual exclusion.

Table 1 summarises the results of the experimental evaluation of our genetic algorithm. First, we report the size of the specification, i.e., the number of goals and domain properties, and the size of the initial population (I.P.) generated from such a specification. For each case study, we ran the algorithm 10 times, with a limit of 50 generations, i.e., 50 evolutions of the genetic algorithm population. Notice that we distinguish between the number of runs in which the genetic algorithm succeeded by identifying at least one boundary condition, and the number of runs in which the algorithm did not identify any boundary condition. For all of the successful runs, we report the minimum, maximum and average number of generations, and the corresponding time in seconds, required for learning the boundary condition. In particular, we focus on the cost of computing the first solution (the number of generations and time – in seconds – required to get a suitable boundary condition), and the cost of computing the "best" solution: notice that the algorithm continues running for 50 iterations, trying to optimise the boundary conditions collected so far, e.g., by making them more compact.

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

Table 1: Evaluation of our Genetic Algorithm for Identifying Boundary Conditions

Case Study				BCs successfully found													No BC found	
					First Solution						Best Solution						–	
					min		**max**		**avg**		**min**		**max**		**avg**		–	
Spec. Name	#Dom	#Goals	I.P.	#Runs	Gen	time	Gen	time	Gen	time	Gen	time	Gen	time	Gen	time	#Runs	avg time
RRCS	2	2	40	8	7	2	11	4	9	2	9	2	50	20	36	17	2	28
MinePump	1	2	34	10	3	1	17	12	6	2	3	1	38	24	18	7	0	0
ATM	1	2	32	9	1	0	18	8	6	2	2	0	48	16	20	7	1	27
Elevator	1	1	22	10	1	0	1	0	1	0	1	0	1	0	1	0	0	0
TCP	0	2	24	9	5	1	39	9	13	3	14	4	50	18	30	10	1	3
Telephone	3	2	42	3	4	4	28	29	14	14	21	29	78	86	30	53	7	44
LAS	0	5	52	3	18	599	37	2355	26	1551	38	3431	47	18403	44	8491	7	4202
AchieveAvoidPattern	1	2	32	10	1	0	8	2	4	1	3	0	26	11	13	5	0	0
RetractionPattern1	0	2	18	10	4	1	38	14	12	4	5	1	50	29	25	17	0	0
RetractionPattern2	0	2	22	8	1	0	25	9	7	2	6	1	39	28	24	16	2	38
Round Robin Arbiter	6	3	94	9	11	32	45	71	22	83	26	32	46	275	38	152	1	170
Simple Arbiter	4	3	128	8	11	245	43	386	28	383	11	245	43	386	29	406	2	1007
Prioritized Arbiter	6	1	84	4	15	257	42	523	30	7428	31	512	50	33687	43	8770	6	1582
Load Balancer	3	8	102	3	15	185	44	9215	34	5253	44	6359	48	11262	46	6578	7	12595
AMBA	6	21	362	6	24	3162	43	16342	30	7100	26	3162	49	7128	33	7541	4	11216
LiftController	7	12	160	5	18	765	47	9690	34	2716	18	2126	47	9690	34	2853	5	22397

As our experiments show, our genetic algorithm can effectively compute boundary conditions for all of the case studies we considered. In the case of small specifications, like the RRCC, Minepump, etc., it can be very efficient, finding the first solution (i.e., a boundary condition) in a few generations. Of course, as the specification becomes more complex, the genetic algorithm needs more iterations to build richer formulas that lead us to boundary conditions, and consequently, it requires much more time. For instance, the worst case we reported is for computing the best solution for the Prioritized Arbiter protocol: our genetic algorithm required 33687 seconds, i.e., more than 9 hours. However, in average the performance of the algorithm is acceptable, considering that it can handle specifications with tens of formulas, that no other related technique can analyse.

5.2 Comparison with Related Techniques

To answer RQ2, we now compare our approach with two related techniques for computing goal conflicts. The first one is a formal approach [53] that requires matching goals against a set of predefined divergence patterns, for which boundary conditions are provided. It provides three different patterns, namely, the Achieve-Avoid pattern, and two versions of the Retraction pattern. The first difference arises from the number of boundary conditions provided by each technique. Table 2 summarises the number of boundary conditions learnt by our genetic algorithm. While patterns are designed to provide only one boundary condition per each divergence pattern, notice that our approach identifies several boundary conditions, that evidence multiple divergence situations that are not contemplated by the patterns. However, it is important to mention that in the case of the Achieve-Avoid and Retraction2 patterns, one boundary condition computed by our approach is equivalent to that provided by the patterns. On the other hand, despite the fact that our approach computes a large number of boundary conditions for the Retraction1 pattern, none of these is comparable (in terms of implication, i.e., either implied by or implying) with that provided by this pattern. This indicates that both techniques complement each other. In fact, one should always apply the patterns when possible, as these provide readable valid boundary conditions, and search for further boundary conditions with other techniques.

The second technique that we consider in the comparison is an automated approach to compute boundary conditions, introduced in [13]. This technique performs a complex logical manipulation of the specification, by using a tableaux-based satisfiability checking algorithm, to identify boundary conditions. It applies to safety and liveness goals, as long as they can be expressed as reachability or response patterns [37]. Table 2 summarises the experimental results of applying this tool to the case studies considered here. The most notable difference is that the tableaux-based technique is not able to analyse all the specifications that our genetic algorithm supports. Basically, the generation of the tableau structure is very expensive, which becomes more noticeable in specifications containing various LTL formulas (in our case studies with larger sets of formulas, most are liveness properties). In these cases, the tableaux-based technique exceeded the timeout, set to 3 hours. The second difference resides again in the number of boundary conditions identified by each approach. While the tableaux-based technique can compute multiple boundary conditions, our genetic algorithm has consistently provided a larger set of divergence situations, that might not have been identified before. As it can be noticed in Table 2, some of the boundary conditions identified by the tableaux-based technique are implied ($\rightarrow$) by some one computed by our genetic algorithm. Others result to be equivalent to some boundary condition computed by our approach. And others resulted to be incomparable, evidencing different kinds of divergences identified by the two approaches. This, again, indicates that the tableaux-based technique can be used, as far as it is able to scale, for complementing the boundary conditions learnt with our approach. The last difference we highlight here is that the tableaux-based approach only applies to safety and liveness goals expressed with the reachability or response patterns, and computes boundary conditions with the general shape $\Diamond\varphi$. In

contrast, our genetic algorithm does not impose any restriction on the LTL formulation of the domain properties and goals, and it does not restrict the shape of the boundary conditions learnt to any particular pattern.

5.3 Scalability and Sensitivity

Basically, a genetic algorithm is a search-based algorithm that is guided to solutions by a fitness function. There are many parameters that may considerably affect the performance of a genetic algorithm: the maximum number of iterations (generations), the size of the population, the maximum size for the chromosomes (in our case that chromosomes have varying lengths), the probability with which certain genetic operator is applied (mutation and crossover rates), etc. An incorrect setting of these parameters may not only affect the performance of the algorithm, it may also affect its effectiveness.

In our algorithm, the maximum number of genes per chromosome is a parameter to which the algorithm is very sensitive. If we select a relatively small size for the chromosomes, the algorithm may be limited to finding boundary conditions that would otherwise be discovered with larger chromosomes, or not discovered at all, if not expressible with such small number of genes. So, this parameter needs to be adapted depending on the specifications involved on each case study. In the case of small specifications, like the MinePump and ATM, 20 genes per chromosome proved to be enough for identifying various boundary conditions. But in the case of larger specifications, as the AMBA or the LiftController case studies, the genetic algorithm required larger chromosomes (e.g., 50 genes per chromosome) to characterise the complex LTL properties involved in the kind of specifications used to express the domain properties and goals.

In order to assess the sensitivity of the genetic algorithm to other parameters, we studied how the effectiveness and efficiency of the algorithm is affected by the progressive variation of these, as it is customary in the context of genetic algorithms. We focused on two parameters, namely, the size of the population maintained per iteration, and the probability with which the genetic operators are applied. On one hand, as it was expected, as the size of the population is increased, we notice that the effectiveness of our genetic algorithm is increased as well. Of course, the efficiency is affected too, since the algorithm has more candidate solutions to which apply the genetic operators and evaluate the fitness function. On the other hand, we observed that the effectiveness and the efficiency of the algorithm do not seem to be affected by the rate of the crossover operator. It is not the same for the mutation operator; as we vary the mutation rate, the effectiveness of the algorithm is affected, but not its efficiency. For instance, in the Telephone case study, with a mutation rate set in 10%, the algorithm was able to find boundary conditions in 5 runs out of 10, i.e., it had an effectiveness of 50%, a 20% more than that reported in Table 1. Thus, despite the fact that our case studies show that our genetic algorithm scales to specifications that cannot be handled by related approaches, we believe that the performance of the algorithm can still be significantly improved by appropriate parameter setting. A more exhaustive experimental evaluation is required to try to identify different classes of problems, and establish well suited configurations of our genetic algorithm in these classes.

5.4 Applicability and Usability

Goal-conflict analysis is typically driven by the identify-assess-control cycle, aimed at identifying, assessing and resolving conflicts that may obstruct the satisfaction of the goals. In particular, the assessment step is concerned with evaluating how likely the identified conflicts are, and how likely and severe are their consequences. The identified conflicts whose likelihood deems them critical, have to be resolved by providing appropriate countermeasures. Notice that for some of the case studies, e.g., the LAS and the round robin arbiter, our genetic algorithm identifies more than one hundred boundary conditions in some runs. Situations like this may make the assessment and control steps very expensive, and even impractical. In order to provide the engineer with an acceptable number of conflicts to be analysed, once the genetic algorithm finished, we perform a number of SAT checks, to attempt to reduce the set of boundary conditions to a smaller set of "more general" ones. Formally, if BC_1 implies BC_2, we say that BC_2 is more general, or weaker, than BC_2. This implication can be checked by using the LTL SAT solver: $BC_1 \wedge \neg BC_2$ is unsatisfiable when BC_1 implies BC_2. Table 3 reports, for each case study, the number of more general BCs. Notice that the number of BCs to be analysed by the engineer can be considerably reduced.

Let us now argue about the usefulness of the computed boundary conditions. Consider the MinePump example, the system that controls a pump in a mine, whose main goal is to avoid a flooding in the mine. The system can detect when the level of the water is high (hw) and when there is methane in the environment (m), since switching on the pump in the presence of methane may produce an explosion. The proposition po is used to indicate that the pump is on. Assume now that the we would like to synthesise a controller that satisfies the following specification:

Domain: *PumpEffect*
InformalDef: If the pump is on, the level of water decreases in at most two time units.
FormalDef: $\Box(\Box_{\leq 2}(po) \rightarrow \Diamond_{\leq 2}(\neg hw))$

Goal: *NoExplosion*
InformalDef: The pump should be off when methane is detected.
FormalDef: $\Box(m \rightarrow \bigcirc(\neg po))$

Goal: *NoFlooding*
InformalDef: The pump should be on when the water level is above the high threshold.
FormalDef: $\Box(hw \rightarrow \bigcirc(po))$

One of the boundary conditions identified by our approach is $BC_1 : \Diamond(m \wedge hw)$, which coincides with the one manually identified in [33], and automatically discovered by the tableaux-based approach [13]. In order to get rid of this conflict, [33] proposes to refine the goal *NoFlooding*, by weakening it, requiring to switch on the pump when the level of water is high and no methane is present in the environment. Thus, *NoFlooding'*: $\Box(hw \wedge \neg \mathbf{m} \rightarrow \bigcirc(po))$. Despite the fact that this refinement removes the boundary condition BC_1, our genetic algorithm still computes 43 additional boundary conditions on the refined specification (3 of them are "more general"), that would demand further attention and subsequent

Table 2: Comparison between our Genetic Algorithm and the Tableaux-based technique

	Genetic Approach - #BCs						Tableaux Approach		Relation			
Case Study	**min**	time	**max**	time	**avg**	time	#BCs	time	$\rightarrow$	$\leftarrow$	$\equiv$	$\not\equiv$
RRCS	5	14	21	23	16	22	1	1	1	0	0	0
MinePump	5	4	53	18	18	9	2	9	0	0	1	1
ATM	4	9	20	17	10	10	4	0	0	0	0	4
Elevator	3	3	17	5	7	3	1	0	1	0	0	0
TCP	4	16	12	20	8	15	2	1	0	0	0	2
Telephone	9	40	48	86	24	66	1	5	0	0	0	1
LAS	46	3729	129	20370	84	9349	1	5	1	0	0	0
AchieveAvoidPattern	12	8	38	11	21	10	4	2	2	0	0	2
RetractionPattern1	18	30	39	42	27	39	1	0	1	0	0	0
RetractionPattern2	11	14	38	44	22	36	1	0	0	0	1	0
Round Robin Arbiter	4	41	103	226	37	174	TIMEOUT					
Simple Arbiter	1	165	44	892	15	1704	TIMEOUT					
Prioritized Arbiter	9	687	19	33751	13	8893	TIMEOUT					
Load Balancer	2	6885	4	13410	3	7565	TIMEOUT					
AMBA	1	38999	7	12286	2	13404	TIMEOUT					
LiftController	1	9302	7	14963	3	15531	TIMEOUT					

Table 3: Number of weakest boundary conditions

Case Study	#BCs	#weakest BCs
MinePump	53	9
ATM	20	5
Elevator	17	3
RRCS	21	3
TCP	12	4
Telephone	48	7
LAS	129	8
AchievePattern	38	9
RetractionPattern1	39	2
RetractionPattern2	38	1
Round Robin Arbiter Unreal 1	103	10
AMBA Unreal 1	7	6
LiftController	7	5
Simple Arbiter Unreal 1	44	6
Load Balancer Unreal 1	4	2
Prioritized Arbiter Unreal 1	19	4

refinements. On the other hand, the tableaux-based approach does not identify any conflict in the refined specification.

In addition to the mentioned use of boundary conditions, other application contexts may be explored, e.g., in synthesis settings. The problem of synthesis consists of automatically producing, from a given specification, an operational model, usually called the *controller*, that by interacting with the environment in which it is executed, it allows it to satisfy a corresponding specification. LTL has been widely used as the specification language in the context of automated synthesis [14, 31, 39]; both the environment and the properties to be satisfied by the controller are captured in many cases in terms of LTL assertions. Typically, a synthesis tool has two possible outputs: that the specification is *realisable*, and a controller is returned; or that the specification is *unrealisable*, meaning that it is not possible to build a controller to guarantee the goals (i.e., the environment always has an strategy to violate them). Unfortunately, when the specification is not realisable, synthesis tools in general do not provide useful feedback to help the user understand why his/her specification is unrealisable. Boundary conditions can be used as declarative sentences useful for diagnosing unrealisable specifications.

For instance, if we consider our previous specification for the Mine Pump Controller, and we ask some synthesis tool, like Ratsy [7], if it is possible to build a controller that satisfies the specified goals, we will get as an answer that the specification is unrealisable. Recall that two boundary conditions computed for this specification were $BC_1 : \Diamond(m \wedge hw)$ and $BC_2 : \Diamond(hw \wedge \neg m \wedge po \wedge \bigcirc(\neg hw \wedge \neg po \vee hw \wedge (m \vee \neg po)))$. These formulas give us information of some admissible behaviours of the system, that lead us to violating the goals. So, if the controller is able to avoid reaching a boundary condition, then such condition is not the reason of the unrealisability of the specification. But, if the controller cannot avoid reaching the boundary condition, then it means that the environment always have a winning strategy to force the controller to reach the boundary condition. In that case, such a BC could be thought of as an explanation of why the controller cannot satisfy the goals, i.e., an explanation of unrealisability of the specification. Returning to the example, if we use Ratsy to check if it is possible to build a controller that avoids boundary condition BC_2 (i.e., we try to synthesize a controller for the goal $\neg BC_2$), then Ratsy will answer that such a goal is realisable. However, if we try to synthesise a controller for goal $\neg BC_1$, Ratsy will answer that such a goal is unrealisable, meaning that it is not possible for the controller to avoid reaching BC_1. Thus, BC_1 can be used to explain why the controller cannot guarantee both goals *NoExplosion* and *NoFlooding* at the same time.

This is a promising application of boundary conditions as explanations of synthesis unrealisability, which needs to be further investigated, and opens a line of future work.

6 RELATED WORK

Inconsistency management has been the focus of many recent works, most of them on the informal or semi-formal side [25, 26, 29, 30]. How to deal with inconsistencies in requirements specifications has also been the focus of several studies on the formal side [16, 17, 24, 43]. From the point of view of qualitative analyses, some works focus on identifying contradictory low-level requirements and computing the degree to which goals are satisfied or denied by them, e.g., [22, 42]. In general, these approaches incorporate the notion of non-functional requirements, and study their relation with the behavioural requirements.

The technique presented in [44] uses abduction for generating explanations for strong inconsistencies, i.e., specifications that are unsatisfiable. Our approach focuses on identifying divergences, that are a weaker form of inconsistency. The technique in [24] searches for inconsistencies between conditional scenarios that describe desired behaviours of the system to be synthesised. Conflicts between non-functional requirements have also been studied, for instance, in [28, 35, 36]. For the resolution of conflicts, [41] makes use of argumentation patterns to elicit, compose and relate stakeholders beliefs. However, it assumes that conflicts have already been elicited, in contrast to our approach that concentrates in identifying goal conflicts. In [49] a methodology is proposed to guarantee that specifications are consistent by construction, eliminating the need for detecting inconsistencies.

As we mentioned in Section 5.4, the problem of detecting inconsistencies in requirements specifications is related to that of realisability of specifications [11, 45]. We believe that discovered boundary conditions can be used for diagnosing unrealisable specifications, providing useful information for the user to understand the cause of the unrealisability. This opens a line of future work, as a deeper exploration of this use of boundary conditions is required. Moreover, it is also somewhat related to the problem of detecting overconstrained specifications [48, 50], e.g., by extracting a core set of assertions that cause an inconsistency in an Alloy model, or by providing some test that identifies a missing behaviour in the model. Our method however attempts to find an explanation (i.e., a boundary condition) that would lead to such inconsistencies.

In the context of the goal-oriented requirements engineering methodologies, obstacles [55] and conflicts [53] have been presented as an abstraction of risks in requirements specifications. Recently, various works [4, 8–10, 55] have been proposed to assist the engineers during the different phases of obstacle analysis. The technique presented in [4] combines model checking and machine learning to automatically generate a set of obstacle conditions with respect to a set of goals and domain properties expressed in LTL. However, obstacles are a particular kind of goal conflict; these are conditions that only affect the satisfaction of one goal. Then, one important limitation of these approaches is that they are ineffective in situations that arise when multiple goals are conflicting.

Various works have also been proposed in order to assist engineers in other phases of the identify-assess-control cycle of conflict analysis, the assessment phase in particular. Some of these techniques, [12] in particular, apply to formal LTL specifications, as in our case, and apply SAT-based mechanisms as well. However, therein the focus in the estimation of goal conflict likelihood, contrary to the aim of the current paper, on boundary condition discovery. Our comparison in Section 5.2 has been limited to techniques specifically targeting goal conflict identification. As already mentioned, [53] introduces the concept of goal conflict, and the pattern-based technique for goal conflict identification. This technique imposes syntactical restrictions on the goal specifications, that seriously limits its applicability; moreover, the patterns are designed to provide only one boundary condition, in cases where more than one boundary condition may exist. Section 5.2 shows that our genetic algorithm is more general, in the sense that it does not impose any restriction of the LTL requirements, and in contrast to the patterns, is able to compute multiple boundary conditions. Moreover, we showed that both approaches might be simultaneously used, when patterns apply, as some boundary conditions are identified by one of the approaches, but not the other.

Another related work is that presented in [13], where boundary conditions are automatically computed using a tableaux-based LTL satisfiability procedure. It applies to safety and liveness goals, as long as these can be captured as the progress and response patterns. This approach is strongly tied to a complex logical algorithm to generate the tableau for the specification (it is an essential part of the approach), which may exhibit scaliability issues. Section 5.2 shows that, for various specifications with larger number of goals and domain properties, the tableau-based mechanism has performance issues. This makes the technique applicable only to relatively small specifications. On the other hand, the genetic algorithm presented in this paper does not impose any restriction on the LTL formulation of the domain properties and the goals, as shows a significantly better performance than the tableaux-based approach; it incorporates a modern LTL satisfiability checker, and is able to efficiently scale to LTL specifications that other approaches are unable to deal with.

7 CONCLUSION AND FUTURE WORK

The identification of inconsistencies during the early phases of requirements engineering is of most importance. It helps in avoiding costly software repairs, and also supports systematic requirements elicitation and verification activities. In this paper we presented a novel approach for identifying goal conflicts in goal oriented LTL requirements specifications, and is based on a genetic algorithm that automatically discovers boundary conditions. The genetic algorithm was designed in a way that allows it to consider arbitrary LTL formulas as boundary condition candidates, and attempts to produce general and compact boundary conditions. Our experiments, based on a large set of case studies, showed that our approach outperforms related techniques, by producing boundary conditions that previous approaches were unable to identify, and being capable of analysing specifications that are beyond the scope of other techniques for boundary condition computation.

Our approach opens various lines for future work. We plan to explore how different parameters affecting the technique's effectiveness can be appropriately set based on characteristics of the analysed specification. We are also exploring the use of alternative fitness functions, based on other logical mechanisms like model counting, to guide the search. Finally, we are studying applications of computed boundary conditions, in particular for unrealisability diagnosis in the context of automated synthesis.

REFERENCES

[1] Java genetic algorithms package (jgap). http://jgap.sourceforge.net.
[2] The reactive synthesis competition. www.syntcomp.org.
[3] Synthesis competition repository. https://bitbucket.org/swenjacobs/syntcomp/.
[4] Dalal Alrajeh, Jeff Kramer, Axel van Lamsweerde, Alessandra Russo, and Sebastián Uchitel. Generating obstacle conditions for requirements completeness. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 705–715, 2012.
[5] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Counter-strategy guided refinement of GR(1) temporal logic specifications. *CoRR*, abs/1308.4113, 2013.
[6] Adrian Beer, Stephan Heidinger, Uwe Kühne, Florian Leitner-Fischer, and Stefan Leue. Symbolic causality checking using bounded model checking. In *Proc. of the 22nd Intl. Sym. on Model Checking Software*, pages 203–221, 2015.
[7] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. RATSY - A new requirements analysis tool with synthesis. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 425–429. Springer, 2010.
[8] Antoine Cailliau and Axel van Lamsweerde. A probabilistic framework for goal-oriented risk analysis. In *2012 20th IEEE International Requirements Engineering Conference (RE), Chicago, IL, USA, September 24-28, 2012*, pages 201–210, 2012.
[9] Antoine Cailliau and Axel van Lamsweerde. Integrating exception handling in goal models. In *IEEE 22nd International Requirements Engineering Conference, RE 2014, Karlskrona, Sweden, August 25-29, 2014*, pages 43–52, 2014.
[10] Antoine Cailliau and Axel van Lamsweerde. Handling knowledge uncertainty in risk-based requirements engineering. In *23rd IEEE International Requirements Engineering Conference, RE 2015, Ottawa, ON, Canada, August 24-28, 2015*, pages 106–115, 2015.
[11] A. Cimatti, M. Roveri, V. Schuppan, and A. Tchaltsev. Diagnostic information for realizability. In *Proc. of the 9th Intl. Conf. on Verification, Model Checking, and Abstract Interpretation*, pages 52–67, 2008.
[12] Renzo Degiovanni, Pablo F. Castro, Marcelo Arroyo, Marcelo Ruiz, Nazareno Aguirre, and Marcelo F. Frias. Goal-conflict likelihood assessment based on model counting. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1125–1135, 2018.
[13] Renzo Degiovanni, Nicolás Ricci, Dalal Alrajeh, Pablo F. Castro, and Nazareno Aguirre. Goal-conflict detection based on temporal satisfiability checking. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 507–518, 2016.
[14] Nicolás Roque D'Ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. Synthesis of live behaviour models. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 77–86, New York, NY, USA, 2010. ACM.
[15] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.
[16] Christian Ellen, Sven Sieverding, and Hardi Hungar. Detecting consistencies and inconsistencies of pattern-based functional requirements. In *Proc. of the 19th Intl. Conf. on Formal Methods for Industrial Critical Systems*, pages 155–169, 2014.
[17] Neil A. Ernst, Alexander Borgida, John Mylopoulos, and Ivan J. Jureta. Agile requirements evolution via paraconsistent reasoning. In *Proc. of the 24th Intl. Conf. on Advanced Information Systems Engineering*, pages 382–397, 2012.
[18] Amy P. Felty and Kedar S. Namjoshi. Feature specification and automated conflict detection. *ACM TOSEM*, 12(1):3–27, 2003.
[19] A. Finkelstein and J. Dowell. A comedy of errors: The london ambulance service case study. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pages 2–, Washington, DC, USA, 1996. IEEE Computer Society.
[20] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.
[21] Dimitra Giannakopoulou and Flavio Lerda. From states to transitions: Improving translation of LTL formulae to büchi automata. In *Formal Techniques for Networked and Distributed Systems - FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 11-14, 2002, Proceedings*, pages 308–326, 2002.
[22] Paolo Giorgini, John Mylopoulos, and Roberto Sebastiani. Goal-oriented requirements analysis and reasoning in the tropos methodology. *Engineering Applications of Artificial Intelligence*, 18(2):159 – 171, 2005.
[23] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.
[24] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Formal Methods in Software and Systems Modeling: Essays Dedicated to Hartmut Ehrig on the Occasion of His 60th Birthday*, pages 309–324, 2005.
[25] J.H. Hausmann, R. Heckel, and G. Taentzer. Detection of conflicting functional requirements in a use case-driven approach. In *ICSE*, pages 105–115, 2002.
[26] Sebastian J.I. Herzig and Christiaan J.J. Paredis. A conceptual basis for inconsistency management in model-based systems engineering. *Procedia CIRP*, 21:52 – 57, 2014.
[27] Pankaj Jalote. *An Integrated Approach to Software Engineering*. Texts in Computer Science. Springer, 2005.
[28] 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 *Proc. of the 18th IEEE International Requirements Engineering Conference*, pages 115–124, 2010.
[29] M. Kamalrudin. Automated software tool support for checking the inconsistency of requirements. In *ASE*, pages 693–697, 2009.
[30] Massila Kamalrudin, John Hosking, and John Grundy. Improving requirements quality using essential use case interaction patterns. In *ICSE*, pages 531–540, 2011.
[31] Uri Klein, Nir Piterman, and Amir Pnueli. Effective synthesis of asynchronous systems from GR(1) specifications. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 283–298, 2012.
[32] J. Kramer, J. Magee, M. Sloman, and A. Lister. CONIC: an integrated approach to distributed computer control systems. *Computers and Digital Techniques, IEE Proceedings E*, 130(1):1+, 1983.
[33] Emmanuel Letier. Reasoning about agents in goal-oriented requirements engineering, 2001.
[34] Jianwen Li, Shufang Zhu, Geguang Pu, and Moshe Y. Vardi. Sat-based explicit LTL reasoning. *CoRR*, abs/1507.02519, 2015.
[35] C. L. Liu. Ontology-based conflict analysis method in non-functional requirements. In *Proc. of the 9th IEEE/ACIS Intl. Conf. on Computer and Information Science*, pages 491–496, 2010.
[36] Dewi Mairiza and Didar Zowghi. Constructing a catalogue of conflicts among non-functional requirements. In *Proc. of the Intl. Conf. Evaluation of Novel Approaches to Software Engineering*, pages 31–44, 2011.
[37] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.
[38] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.
[39] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. In Dexter Kozen, editor, *Logics of Programs*, pages 253–281, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.
[40] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 1996.
[41] P.K. Murukannaiah, A.K. Kalia, P.R. Telangy, and M.P. Singh. Resolving goal conflicts via argumentation-based analysis of competing hypotheses. In *Proc. 23rd IEEE Int. Requirements Engineering Conf.*, pages 156–165, 2015.
[42] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Trans. Softw. Eng.*, 18(6):483–497, June 1992.
[43] Tuong Huan Nguyen, Bao Quoc Vo, Markus Lumpe, and John Grundy. KBRE: a framework for knowledge-based requirements engineering. *Software Quality Journal*, 22(1):87–119, 2013.
[44] Bashar Nuseibeh and Alessandra Russo. Using abduction to evolve inconsistent requirements specification. *Australasian Journal of Information Systems*, 6(2), 1999.
[45] Suchismita Roy, Sayantan Das, Prasenjit Basu, Pallab Dasgupta, and P. P. Chakrabarti. Sat based solutions for consistency problems in formal property specifications for open systems. In *CAD*, pages 885–888, 2005.
[46] Kristin Y. Rozier and Moshe Y. Vardi. LTL satisfiability checking. *STTT*, 12(2):123–137, 2010.
[47] Viktor Schuppan. Towards a notion of unsatisfiable and unrealizable cores for ltl. *Sci. Comput. Program.*, 77(7-8):908–939, July 2012.
[48] Ilya Shlyakhter, Robert Seater, Daniel Jackson, Manu Sridharan, and Mana Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *ASE*, pages 94–105, 2003.
[49] Monique Snoeck, Cindy Michiels, and Guido Dedene. Consistency by construction: The case of merode. In *Proc. of the ER Workshop on Conceptual Modeling for Novel Application Domains*, pages 105–117, 2003.
[50] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. Automated test generation and mutation testing for alloy. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 264–275, 2017.
[51] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of behavioral models from scenarios. *IEEE Trans. Software Eng.*, 29(2):99–115, 2003.
[52] Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.
[53] Axel van Lamsweerde, Robert Darimont, and Emmanuel Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. Software Eng.*, 24(11):908–926, 1998.
[54] Axel van Lamsweerde and Emmanuel Letier. Integrating obstacles in goal-driven requirements engineering. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 53–62, Washington, DC, USA, 1998. IEEE Computer Society.

[55] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Trans. Softw. Eng.*, 26(10):978–1005, October 2000.