Goal-Conflict Likelihood Assessment based on Model Counting

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

Pablo Castro
Universidad Nacional de Río Cuarto
and CONICET, Argentina
pcastro@dc.exa.unrc.edu.ar

Marcelo Arroyo
Universidad Nacional de Río Cuarto
Argentina
marcelo.arroyo@dc.exa.unrc.edu.ar

Marcelo Ruiz
Universidad Nacional de Río Cuarto
Argentina
mruiz@exa.unrc.edu.ar

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

Marcelo Frias
Instituto Tecnológico de Buenos Aires
and CONICET, Argentina
mfrias@itba.edu.ar

ABSTRACT

In goal-oriented requirements engineering approaches, conflict analysis has been proposed as an abstraction for risk analysis. Intuitively, given a set of expected goals to be achieved by the system-to-be, a conflict represents a subtle situation that makes goals diverge, i.e., not be satisfiable as a whole. 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 expected goals. In particular, the assessment step is concerned with evaluating how likely the identified conflicts are, and how likely and severe are their consequences.

So far, existing assessment approaches restrict their analysis to obstacles (conflicts that prevent the satisfaction of a single goal), and assume that certain probabilistic information on the domain is provided, that needs to be previously elicited from experienced users, statistical data or simulations. In this paper, we present a novel automated approach to assess how likely a conflict is, that applies to general conflicts (not only obstacles) without requiring probabilistic information on the domain. Intuitively, given the LTL formulation of the domain and of a set of goals to be achieved, we compute goal conflicts, and exploit string model counting techniques to estimate the likelihood of the occurrence of the corresponding conflicting situations and the severity in which these affect the satisfaction of the goals. This information can then be used to prioritize conflicts to be resolved, and suggest which goals to drive attention to for refinements.

CCS CONCEPTS

• **Software and its engineering** → **Requirements analysis**; **Risk management**; • **Theory of computation** → *Modal and temporal logics*; Regular languages;

KEYWORDS

Goal Conflicts, Risk Likelihood Assessment, Model Counting

ACM Reference Format:
Renzo Degiovanni, Pablo Castro, Marcelo Arroyo, Marcelo Ruiz, Nazareno Aguirre, and Marcelo Frias. 2018. Goal-Conflict Likelihood Assessment based on Model Counting. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 11 pages.
https://doi.org/10.1145/3180155.3180261

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180261

1 INTRODUCTION

As many errors in software development are due to an incorrect understanding of what the system should do, putting sufficient emphasis in requirements analysis and specification is known to be a major task toward successful software development projects [27]. There exist many reasons for requirements descriptions to be inadequate, including requirements being too ideal (e.g., assuming that some aspect of the environment will behave unrealistically benevolently with the system), requirements being too abstract and thus leaving room for conflicting situations whose treatment goes unspecified, missing some requirements altogether or leaving them implicit, and even having inconsistent requirements that cannot be satisfied as a whole [40]. The requirements analysis phase deals precisely with these issues, and attempts to identify such problems as early as possible, so that appropriate actions can be taken to improve the software requirements prior to development.

A particular family of approaches toward requirements analysis and specification are the so called *goal-oriented requirements methodologies* [14, 45]. In such approaches, requirements are organized around *goals*, prescriptive statements that specify what the system to be developed should do. These goals are subject to a number of activities (that are part of the requirements process); they can be analyzed, decomposed, refined, assigned to agents for their realization, and assessed in a number of different ways, including assessments that evaluate their feasibility, and potential threats to their satisfaction. Indeed, *risk analysis* deals precisely with this last issue, in various ways, including *goal-conflict analysis* [41]. Goal-conflict analysis aims at identifying *conflicts*, i.e., conditions that, when present, make a set of otherwise consistent goals inconsistent, as well as assessing and controlling such conflicts. Once a conflict is identified, the assessment step is concerned with evaluating how *likely* the identified conflict is, and how severe are its consequences. So far, existing assessment approaches [4, 11–13, 43] restrict their

analyses to simpler kind of conflicts, called *obstacles*, and assume that certain probabilistic information on the domain is provided.

While goal-conflict assessment can be naturally thought of as a *quantitative* analysis that uses probabilities of conflicting situations to analyze their impact in the overall requirements description, many times one is unable to come up with the required probabilities for such analyses. The reason is that the required probabilistic information needs to be obtained prior to the goal-conflict analysis from some trusted source, e.g., available statistical data, simulations, or stakeholders whose expertise allows them to estimate these probabilities. In many development projects, especially new developments, such information may be unavailable, in particular in early phases of requirements analysis. Thus, in these situations, important requirements activities such as conflict prioritization and goal refinement, that benefit from goal-conflict analysis, need to be postponed or performed without such information.

In this paper we propose an automated technique for goal-conflict analysis. This technique is novel in two respects, namely, it does not demand probabilistic information, and it applies to general conflicts (as opposed to previous works that are restricted to goal obstacles). The approach is based on automatically computing goal conflicts, capturing these as formulas, and using *model counting* to estimate their corresponding likelihood. More precisely, given an LTL formulation of the domain and of a set of goals to be achieved, we compute goal conflicts, and exploit string model counting techniques to estimate the likelihood of the occurrence of the corresponding conflicting situations and the severity in which these affect the satisfaction of the goals.

Basically, from an LTL formula φ, we can generate a regular expression e_φ representing the prefixes accepted by φ. We can then use a string model counter to compute the number of strings (prefixes) of length k that are recognized by the regular expression e_φ, as an estimation of the number of models that satisfy φ. This tool can compute the number of (finite) traces, for a given bound, that satisfy the domain description, and to calculate how many of these correspond to a particular conflicting situation, as a way of estimating the likelihood of the conflict. This information can then be used to prioritize conflicts to be resolved, and suggest which goals to drive attention to for refinements. We develop a number of case studies, with requirements models taken from the literature, for which we perform automated goal conflict detection, and model counting for their assessment.

The remainder of the paper is organized as follows. Section 2 introduces preliminary concepts necessary in subsequent sections. Section 3 presents an illustrating example, that motivates our approach. Section 4 describes the approach in detail. In Section 5 we evaluate our technique, by analyzing how model counting can be used to estimate the likelihood of conflicting situations, in a number of case studies. Finally, we discuss related work in Section 6, and present our conclusions in Section 7.

2 BACKGROUND

2.1 Goal-Oriented Modelling and Conflict Analysis

Among the various approaches to Requirements Engineering, goal-oriented methodologies, and in particular the one presented in [40], propose organizing the specification of how a system should behave around a set of *goals*. These goals are *prescriptive* statements that the system to be developed is expected to achieve in cooperation with other *agents* (e.g., devices, users, etc, besides the software itself), within a given *domain*. The domain is captured through *domain properties*, *descriptive* statements about the problem world, such as, e.g., natural laws relevant to the system. A *goal model* indicates, through refinements, how higher-level goals can be achieved in terms of simpler ones, eventually simple enough so that they can be assigned to single agents for their fulfillment.

There exist many reasons for requirements descriptions to be inadequate, including requirements being too ideal (e.g., assuming that some aspect of the environment will behave unrealistically benevolently with the system), requirements being too abstract (and thus leaving room for conflicting situations whose treatment is underspecified), missing some requirements altogether (or leaving them implicit), and even having inconsistent requirements that cannot be satisfied as a whole.

The process leading to the right requirements is not straightforward. *Conflict analysis* [40, 42] helps in improving requirements specifications through three main steps: *(i)* identify as many conflicts as possible and relate them to (sets of) goals in the goal model, *(ii)* assess how likely the identified conflicts are, and how likely and severe are their consequences; and *(iii)* resolve those conflicts whose likelihood deems them critical, by providing appropriate countermeasures and, consequently, transforming the goal model.

Conflicts in goal models represent conditions whose occurrence result in the loss of satisfaction of the goals, i.e., that make the goals *diverge*. Formally, a set $G_1, \ldots, G_n$ of goals is said to be *divergent* with respect to a set *Dom* of domain properties iff there exists a *boundary condition BC* such that the following conditions hold [41]:

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

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

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

Intuitively, the above conditions indicate that the boundary condition is consistent with domain and captures a particular combination of circumstances that makes the goals conflicting. That is, when the boundary condition *BC* holds, then the goals cannot be simultaneously satisfied in *Dom* under any circumstances. Boundary conditions represent a very general form of conflict, that in particular subsume *obstacles* [43], a particular instance of conflicts in which *G* only contains one single goal.

There are relevant approaches that can automatically identify boundary conditions for conflicting goals written in Linear-Time Temporal Logic (LTL). The pattern-based approach put forward in [41] supports a limited set of patterns and only produces predetermined formulations of boundary conditions. The approach introduced in [15] automatically produces boundary conditions through the processing of an LTL tableaux for the goals and domain properties. We will use the latter to identify boundary conditions, whose likelihood will be assessed through model counting.

2.2 Linear-Time Temporal Logic

Linear-Time Temporal Logic (LTL) [34] is a logical formalism widely employed to formally state properties of reactive systems. This logic

is used as the specification formalism in various tools for formal analysis, including model checkers [26, 33], trace checkers [7] and theorem provers [10], among others.

LTL assumes that time is *linear*, i.e., each instant is succeeded by a single future instant. The syntax of LTL is formally defined as follows. Given a set AP of propositional variables, LTL formulas are inductively defined using the standard logical connectives and temporal operators $\bigcirc$ and $\mathcal{U}$, by the following rules: *(i)* every $p \in AP$ is an LTL formula, and *(ii)* if f_1 and f_2 are LTL formulas, then so are $\neg f_1$, $f_1 \vee f_2$, $f_1 \wedge f_2$, $\bigcirc f_1$ and $f_1 \mathcal{U} f_2$. We consider the usual definition for the operators $\Box$ (always) and $\Diamond$ (eventually) in terms of $\bigcirc$, $\mathcal{U}$ and logical connectives [34].

2.3 LTL Model Counting

LTL formulas are interpreted over infinite traces of propositional valuations. These traces are typically finitely-represented through *finite-state transition systems*, i.e., they correspond to (infinite) words over a given finite-state transition system. Thus, problems regarding LTL are usually expressed in relation to these finite-state transition systems. A known example is model checking [6], which is defined as the problem of deciding, given a transition system T and an LTL formula φ, whether all executions of T satisfy φ. Similarly, LTL satisfiability can be defined as the problem of deciding, given an LTL formula φ, if there exists a finite-state transition system T with at least one execution that satisfies φ. The *model counting* problem for LTL is then the problem of, given an LTL formula φ, counting how many transition systems satisfy φ. It is not difficult to see that, if an LTL formula φ has a model, then it has an *infinite* number of models (e.g., one can "unfold" a satisfying transition system T any finite number of times, obtaining again a model of φ). Then, an LTL formula can have either an infinite number of models (when it is satisfiable), or no model at all (when it is unsatisfiable).

In order to obtain a more useful value, the LTL model counting problem is usually restricted to *bounded models*. That is, given an LTL formula φ and a bound k, the model counting problem consists of calculating how many models of at most k states exist. Based on known results from the area of *bounded model checking* [9], one can typically restrict the analysis to certain kind of canonical models. In [20], for instance, two different kinds of bounded models for LTL are considered, namely *k-word* models and *k-tree* models. We will concentrate in k-word models.

A word model can be represented by a finite sequence of states, called the *base* of the word, such that the last state has a loop to some previous state. These are called *lasso traces* [20], and Figure 1 shows their general shape. Formally, given a set AP of atomic propositions, a word model σ is a sequence of states $s_0, \ldots, s_{i-1}, (s_i, \ldots, s_k)^\omega$, such that, $s_0, \ldots, s_k \in 2^{AP}$; in such a word model, there is a loop from state k to state i.

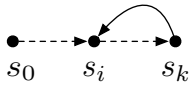

Figure 1: Shape of word models.

The straightforward approach one can use to count the number of word-models is through SAT-based Bounded Model Checking [9]. For a given bound k, LTL formulas can be encoded as propositional formulas [32]. The encoding is such that *each* satisfying valuation of the encoding corresponds to a lasso trace of k states of the LTL formula. Then, a model counter for propositional logic, e.g., `RelSAT` [28], `cachet` [37] and `sharpSAT` [38], can be used for "indirect" LTL (bounded) model counting. An alternative approach, presented in [20], introduces an automata-based algorithm for counting word models of *safety* LTL formulas. It proceeds in two steps: first, it constructs a word automaton that accepts a finite sequence of size k if it is a base for a word model of φ; and second, it uses the word automaton to calculate how many loops are allowed in order to count the number of word models of φ.

As explained in [20], both of the above approaches become impractical for relatively small values of k. On one hand, the reduction to propositional counting leads us to propositional constraints with thousands of variables that cannot be efficiently handled by available model counters. In case of the algorithm in [20], it is linear in k, but double exponential on the size of the LTL formula, and only applies to safety formulas.

2.4 String Model Counting

In order to perform a significant assessment on how likely a goal-conflict is, we need the model counting analysis to scale to relatively large values of k, and thus current LTL model counting technology quickly reaches its limits. However, if we consider counting *word bases* instead of word models, the problem can be very efficiently solved by using string model counting. The problem of string model counting consists of computing the number of strings of a given length k, that satisfy a set of string constraints. For instance, given a regular expression $e = (a|b)^*$, we may need to know how many strings of, say, length 2, satisfy the constraint e (in this case, it is 4: *aa*, *ab*, *ba* and *bb*). In this work, we use the recently introduced string model counter `ABC` [5]. Given a set C of string constraints, `ABC` first constructs an automaton that accepts all the strings satisfying the constraints in C, and then produces a *generating function* that takes a length bound k as input and returns the total number of strings of length k, that are accepted by the automaton. We denote by $\#(C, k)$ the result of evaluating in k the generating function produced from the set of string constraints C. `ABC` checks the satisfiability of C only once, when producing the generating function; after that, we can evaluate the generating function on many different values of k, without having to check for satisfiability again. This is the main feature of `ABC` that allows us to scale to considerably large values of k (e.g., 1000), in seconds. `ABC` supports a wide set of string constraints, but we use only regular expressions.

2.5 Word Base Counting

Figure 1 shows the general shape of a word model. In particular, the sequence of states $s_0, \ldots, s_k$ is the *base* of the word. Notice that the word base can be thought of as a finite string of length k that can be accepted by a regular expression. Then, if we can encode an LTL formula φ into a set C_φ of string constraints, we can use `ABC` to compute $\#(C_\varphi, k)$, the number of word bases of length k satisfying φ. For instance, for the LTL formula $\Box(p \vee q)$, our approach produces the regular expression $(a|b|c)^*$, where characters a, b and c characterise the 3 different possible ways of making $p \vee q$ true. Thus, for $k = 4$, $\#((a|b|c)^*, 4) = 81$, meaning that there are 81 word

bases that can be obtained from word models of at most 4 states. Notice that each string we count may be the base of different word models. For instance, we count the word base *aaaa* only once, but there are 4 different word models with the same base but with different loops (namely, $(aaaa)^\omega$, $a(aaa)^\omega$, $aa(aa)^\omega$ and $aaa(a)^\omega$).

More precisely, given an LTL formula φ, we generate the regular expression e_φ by performing the following encodings: (1) we generate the Büchi automaton B_φ that recognises φ; (2) we generate a Finite State Automaton A_φ from B_φ; and (3) we generate the regular expression e_φ from A_φ. Intuitively, the Büchi automaton B_φ recognises all the word models for formula φ. Recall that Büchi automata only recognise infinite words, since their accepting condition requires that some accepting state has to be visited infinitely often in the word model. However, when we generate the finite automaton A_φ from B_φ, the strings recognised by A_φ are finite (its accepting condition requires only reaching some final state). Because of that, the words recognised by A_φ are the bases of some word model. Finally, we can convert A_φ to a regular expression e_φ that recognises exactly the same language, i.e., the word bases for the LTL formula φ.

Along the process, we apply a minimization algorithm to the Büchi automaton, to produce a regular expression that is more compact and easier to analyse. All the conversions are done with the tools `LamaConv` [2] and `JFLAP` [1]. Our experimental evaluation shows that the translations can be performed very efficiently.

3 MOTIVATING EXAMPLE

Let us consider a simplified version of the Mine Pump Controller (MPC) [30], to illustrate both the problem addressed and our proposed approach. The MPC controls a pump in a mine. It communicates with a sensor that detects when the water level is high, and a sensor to detect the presence of methane in the environment. The propositional variables *hw*, *m* and *po* are employed to represent the facts that the water level has reached a high threshold, that methane is present in the environment, and that the pump is on, respectively. For this problem setting, the following assumption and goals have already been elicited:

Assumption: *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: $\Box(hw \rightarrow \bigcirc(po))$

The above assumption captures the presumed behaviour of the pump actuator in relation to the environment: when the pump is on, then at most in 2 time units the water level should decrease. The goals prescribe how the pump should work when there is methane present in the environment, and when the water level is high, respectively. Notice that while these goals can be simultaneously satisfied (e.g., when the water level is never high or methane is never present in the environment), they become logically inconsistent when, at the same time, the water level is high and methane is present. Such conflicting situations, that as we mentioned are called *boundary conditions*, are many times subtle and difficult to identify, and at the same time their finding is essential for the requirements process. In this case, the mentioned boundary condition is $BC_1 : \Diamond(hw \wedge m)$, and is one of the two boundary conditions automatically computed using the tool presented in [15]. The other boundary condition is less legible: $BC_2 : \Diamond(hw \wedge \neg m \wedge po \wedge \bigcirc(\neg hw \wedge \neg po \vee hw \wedge (m \vee \neg po)))$.

These boundary conditions capture *different* conflicting situations that lead us to violating the goals. To continue with the *identify-assess-control* approach for goal-conflict analysis, we should now attempt to assess how likely these conflicting situations are, and in what degree they affect each of the goals. This information is of course very useful for the engineer, as it can be used for classifying the boundary conditions according to their criticality, and based on which goals are more affected, may suggest actions to the engineer toward resolving the conflicts.

In an ideal situation, we may have statistical information regarding the chances of sensing methane in the environment, or how often the water level reaches the high threshold, enabling us to perform some probabilistic analysis as in [11], even with uncertainties in the elicited values [13]. Our current approach, on the other hand, tries to help the engineer when such information is unavailable.

A typical approach when no probabilistic information is available, is assuming that all events are equally likely, and performing a probabilistic analysis under such hypothesis. For instance, under such an assumption, we can use a quantitative analysis such as *probabilistic model checking* [6], to analyze the likelihood of our identified boundary conditions. This approach is however ineffective: both boundary conditions can be eventually reached with probability 1, and thus are indistinguishable (in terms of severity) by such technique.

As mentioned before, our technique is based on using string model counting to assess the boundary conditions. We can start by computing, for some given bound k, the number of word bases that satisfy our domain properties (*true* for this example) and assumptions (*PumpEffect*), denoted by $\#(PumpEffect, k)$. Notice that by $\#(PumpEffect, k)$ we denote the number of strings of length k that are recognised by the regular expression obtained from *PumpEffect*. Then, we can compute how many of these finite traces have a state where boundary condition BC_1, and (separately) boundary condition BC_2, are reached (i.e., we compute $\#(\{PumpEffect, BC_1\}, k)$ and $\#(\{PumpEffect, BC_2\}, k)$). The quotient between the number of word bases of length k that satisfy BC_1 (resp. BC_2) and the number of all "valid" word bases (i.e., word bases that satisfy the domain properties and assumptions) of length k, give us an estimation of the likelihood of reaching the boundary condition BC_1 (resp. BC_2) in k steps. Table 1 summarizes, for several values of k, the number of word bases that satisfy the assumption, and in addition each of the boundary conditions, for our mine pump model.

This first straightforward analysis already allows us to classify the boundary conditions. Notice that, as the evaluation shows, the chances of reaching BC_1 tends to 0% as the value of k is increased (despite of having good chances initially – 22%); while the chances of reaching BC_2 converges to %6.2. So, we can classify BC_2 as being "more likely" than BC_1 in the long term, and the engineer should

Table 1: Counting word bases of length k for the boundary conditions of the Mine Pump Controller.

	k											
$\#(C_\varphi, k)$	1	2	3	4	5	6	10	20	50	75	100	1000
$\#(PumpEffect, k)$	9	64	448	3136	21888	152320	357462016	9.53E+16	1.81E+42	2.01E+63	2.43E+84	5.17E+843
$\#(\{PumpEffect, BC_1\}, k)$	2	12	54	306	1620	8586	6918210	1.27E+13	7.81E+35	1.12E+54	1.616E+72	9.62E+725
$\%BC_1$	0.22	0.19	0.12	0.1	0.07	0.06	0.02	0.001	0	0	0	0
$\#(\{PumpEffect, BC_2\}, k)$	0	5	28	192	1376	9472	22265856	5.94+15	1.12+41	1.31E+62	1.5E+83	3.76E+841
$\%BC_2$	0	0.08	0.062	0.061	0.063	0.062	0.062	0.062	0.062	0.062	0.062	0.062

prioritise BC_2 in the search for mechanisms that would allow us to reduce the chances of reaching BC_2.

Our simple model counting approach also enables us to perform some more detailed analyses. We know that when a boundary condition is reached, then the goals are violated. Despite the fact that we have already estimated that BC_2 is more likely than BC_1, we do not know *how much* each boundary condition affects the satisfaction of every particular goal. To analyze this, we can first compute $\#(\{PumpEffect, NoExplosion\}, k)$, and then calculate how many of these are consistent with each boundary condition, by computing:

$$\#(\{PumpEffect, NoExplosion, BC_1\}, k),$$
$$\#(\{PumpEffect, NoExplosion, BC_2\}, k).$$

The quotient between the number of word bases of length k that satisfy both the goal *NoExplosion* and the boundary conditions, and the number of all word bases satisfying only the goal, give us useful information to assess to what extent each of BC_1 and BC_2 affect the satisfaction of goal *NoExplosion* (the same counting problem is applied for goal *NoFlooding*). Table 2 shows how the boundary conditions identified for our MPC example affect the satisfaction of each particular goal.

Various issues become immediately apparent, as we analyze Table 2. Regarding BC_1 (i.e., $\Diamond(hw \wedge m)$), clearly this conflicting situation affects the satisfaction of both goals to the same extent, since the 0% that is observed as k is increased indicates that no trace reaching BC_1 will be able to satisfy the goals. Regarding BC_2, it is clear that it seriously affects both goals, but it affects to a greater extent to goal *NoExplosion*: there is no way in which *NoExplosion* can be satisfied when BC_2 holds. Depending on the value that the engineer assigns to each goal, this information may suggest priorities in the actions to take to improve the model. For instance, if guaranteeing *NoExplosion* is mandatory, then dealing with BC_2 is critical, as it forbids the satisfaction of this goal.

4 THE APPROACH

Our approach for goal-conflict likelihood assessment receives a goal-oriented requirements specification, composed of LTL formulas capturing the domain properties *Dom*, as well as goals $G = \{G_1, \ldots, G_n\}$. The process starts by identifying goal conflicts. If no conflict is detected, there is nothing else to be done. On the other hand, when some conflicts are detected, these are automatically produced as a set $BC = \{BC_1, \ldots, BC_m\}$ of *boundary conditions*, characterizing different divergent situations between the goals in the domain. Assuming that we have *no statistical information* regarding the likelihood of events in the specification, we will estimate the likelihood of the identified boundary conditions, and to what extent these affect the goals, through model counting, as put forward in the previous section. These estimations can be exploited by the engineer to prioritize the treatment of conflicts, and to drive attention to certain aspects of the specification, for its modification or refinement. This of course is a task that requires ingenuity, and the specific corrective actions to be taken to handle conflicts cannot be derived solely from our likelihood estimation; we nevertheless provide, for the sake of example, some requirements modification and refinement cases, based on our analysis. Figure 2 depicts the workflow followed by our approach.

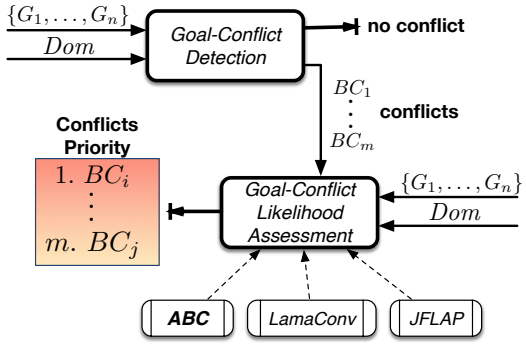

Figure 2: Overview of the approach.

Goal-Conflict Detection. Our approach starts by identifying goal conflicts. While any approach to goal conflict identification would fit our technique, including manual ones, we employ an automated technique to compute boundary conditions, introduced in [15]. The reasons are many-fold. Firstly, having an automated goal-conflict detection approach allows us to start from goals and domain properties, and automate the whole process of prioritizing conflicts, and assessing the effect of conflicts on goals satisfaction. Secondly, the approach is more general than other techniques for identifying boundary conditions, notably [41], that is restricted to certain patterns. Thirdly, boundary conditions are more general, as conflict descriptions, than other kinds of conflicts such as obstacles, thus broadening the scope of our analysis.

Goal-conflict detection is not the main issue in this paper, and we refer the reader to [15] for details on the tableaux-based goal-conflict computation approach. We use the tool in a "black-box" manner. We will simply remark that the tool is able to identify boundary conditions from safety as well as liveness goals, as long as the latter are expressed as reachability or response patterns [34]. So, we assume that the goals and domain properties are of these kinds. Boundary conditions are always of the form $\Diamond\varphi$, where φ is a state property if the goals are safety goals, and is a persistence formula $\Box\varphi'$, when the goals are liveness goals.

Table 2: Loss of Goal Satisfaction for the Mine Pump Controller, in k-bounded word bases.

$\#(C_\varphi, k)$	k											
	1	2	3	4	5	6	10	20	50	75	100	1000
$\#(\{PumpEffect, NoExplosion\}, k)$	9	48	224	1040	4864	22400	9990144	4.17E+13	3.05E+33	1.09E+50	3.89E+66	3.16E+662
$\#(\{PumpEffect, NoExplosion, BC_1\}, k)$	0	2	11	62	332	1580	667624	1.37E+12	6.82E+30	2.52E+46	9.32E+61	2.58E+622
$\%BC_1$	0	0.042	0.049	0.059	0.068	0.070	0.067	0.033	0.002	0	0	0
$\#(\{PumpEffect, NoExplosion, BC_2\}, k)$	0	0	2	8	52	266	146568	6.73E+11	4.95E+31	1.76E+48	6.31E+64	5.12E+660
$\%BC_2$	0	0	0.009	0.008	0.01	0.011	0.014	0.016	0.016	0.016	0.016	0.016
$\#(\{PumpEffect, NoFlooding\}, k)$	9	48	248	1280	6504	32984	21633944	2.37E+14	3.14E+35	1.25E+53	5.02E+70	2.27+E704
$\#(\{PumpEffect, NoFlooding, BC_1\}, k)$	0	2	16	104	546	2722	1280356	3.92E+12	8.87E+31	1.19E+48	1.59E+64	6.25E+644
$\%BC_1$	0	0.042	0.065	0.081	0.084	0.083	0.059	0.016	0	0	0	0
$\#(\{PumpEffect, NoFlooding, BC_2\}, k)$	0	4	12	80	400	2084	1407600	1.55E+13	2.06E+34	8,23E+51	3.28E+69	1.49E+703
$\%BC_2$	0	0.083	0.048	0.062	0.061	0.063	0.065	0.065	0.065	0.065	0.065	0.065

Goal-Conflict Likelihood Assessment. This phase, concerned with evaluating how likely the identified conflicts are and how severe are their consequences, is the main phase of our approach, where model counting takes place. The input for this phase are the LTL formulation *Dom* of the domain, the set $\{G_1, \ldots, G_n\}$ of goals, and the set $\{BC_1, \ldots, BC_m\}$ of boundary conditions previously identified.

As explained in Section 2, given an LTL formula φ, we can generate a regular expression e_φ, such that e_φ is satisfiable iff there exists a word-model whose word-base is recognised by e_φ. Then, we can feed this regular expression to a string model counter, in our case ABC, to compute the number of strings that satisfy e_φ. The obtained number of instances indirectly represents the number of word bases, that are part of some word model that satisfies the LTL formula φ. We denote this last notion by $\#(e_\varphi, k)$, or, equivalently, by $\#(\varphi, k)$. Recall that when the string constraint e_φ is satisfiable, ABC produces a *generating function* that can be used for solving the counting problem for different values of k, without the need of rechecking the satisfiability of e_φ. This feature is of utmost importance to scaling the analysis to sufficiently large values of k, allowing us to perform a meaningful assessment on how likely a boundary condition is.

Recall that every boundary condition $BC_i \in BC$ has the canonical shape $\Diamond\varphi_i$ (see above). Thus, the canonical shape of the regular expression generated from BC_i is:

$$e_{\Diamond\varphi_i} = e^*_{\neg\varphi_i}\ e_{\varphi_i}\ \texttt{alph}^*$$

where alph represents any character from the alphabet of the regular expression. Intuitively, the regular expression $e_{\Diamond\varphi_i}$ recognises strings that initially have a prefix in which φ_i does not hold ($e^*_{\neg\varphi_i}$), then the eventuality φ_i is fulfilled (e_{φ_i}), and from that point on any character is recognised. Given $e_{\Diamond\varphi_i}$, by removing the suffix $\texttt{alph}^*$, we can obtain a regular expression $e'_{\Diamond\varphi_i}$ that recognises the word bases of the LTL formula $\Diamond\varphi_i$:

$$e'_{\Diamond\varphi_i} = e^*_{\neg\varphi_i}\ e_{\varphi_i}$$

Notice that $e'_{\Diamond\varphi_i}$ forces the boundary condition φ_i to hold exactly in the last state of any string recognised by such a regular expression.

Thus, for each boundary condition $BC_i \in BC$, we start by generating the regular expression that characterises the domain properties *Dom*, and the regular expression $e'_{\Diamond\varphi_i}$ as we just explained. Then, we perform the string model counting process for each one of the following string constraints:

$$(A)\#(Dom, k) \qquad (B)\#(\{Dom, e'_{\Diamond\varphi_i}\}, k)$$

Intuitively, the model counting problem *(A)* gives us the number of all possible word bases, of length k, that are part of some word model that satisfies the domain properties *Dom*. On the other hand, the model counting problem *(B)* gives us how many of the word bases computed in *(A)*, reach the boundary condition BC_i, for the first time, *exactly* in the k-th state. The reason for counting in this way, as opposed to directly counting the traces that reach the boundary condition BC_i in *any* state, has to do with avoiding to count several times what would be, essentially, the same violation. For instance, assume an alphabet $\{p, q\}$ of two propositional letters, the boundary condition being represented by $\Diamond(p \wedge q)$, and a k value of 3. If $p \wedge q$ holds in the first state, then no matter what happens later on in the trace, they will all be violations (a known fact of safety properties as interpreted in [31] and formalized in [3]). Then, we will be counting 16 different violations (4 different valuations for the second state, times 4 different violations for the third state), that are, in essence, all the same one. A more precise way of counting violations is, as we propose, taking into account the traces that reach the property of interest exactly in the k-th state, as we do (or, even, calculating the sum of violations from 1 up to k, where, in each case, we stipulate that the property must be first reached exactly at the last state).

Using the results of model counting as shown in *(A)* and *(B)*, we can compute the quotient *(B)*/*(A)*, that represents the *likelihood* of reaching, for the first time, the boundary condition BC_i in exactly k steps. In order to assess how this probability is modified as the trace length is increased, we perform the above model counting iteratively for increasingly large values of k, until some predefined maximum value N is reached (or some other predefined stopping criterion is reached, e.g, a timeout). These computed likelihoods can be used to generate a progression of how the number of word bases reaching BC_i relate to the number of word bases satisfying domain properties. These progressions generated for each $BC_i \in BC$ should be analysed by the engineer, to determine which boundary conditions are more likely, and prioritize these for resolution.

The selection of the maximum value N should be large enough to allow the model counting process to converge the likelihood computation to some average value. In our experimental evaluation we use 1000 as the maximum value for k. If the progressions computed converge to some constant value, then the average of the progressions can be used as a quantifiable value to classify which boundary condition is more likely. For instance, in the Mine Pump Controller, the progressions for BC_1 converges to 0%, while the progressions of $\%BC_2$ converges to 6.2%. This indicates that BC_2

has more chances to be reached than BC_1 and the engineer should prioritise $BC2$ in the search for mechanisms that would allow him to reduce the chances of reaching BC2.

Goal-Conflict Severity Assessment. When a boundary condition is reached, it is not possible to satisfy, simultaneously, all the goals. Our above quantitative assessment based on model counting only tells us how many violations of length k we have, over the total number of possible "valid" executions (i.e., executions where domain properties hold) of length k. In order to assess the impact of a boundary condition on a specific goal, we need to perform a different analysis. Essentially, we would like to evaluate to what extent a boundary condition contradicts a specific goal. In order to analyze this issue, for each boundary condition $BC_i \in BC$ and goal $G_j \in G$, we perform the following model counting tasks:

$$(C)\#(\{Dom, G_j\}, k) \qquad (D)\#(\{Dom, G_j, e'_{\Diamond\varphi_i}\}, k)$$

Intuitively, counting problem *(C)* counts the number of word bases of length k that satisfy both the domain and goal G_j. On the other hand, model counting problem *(D)* counts how many of the word bases that satisfy Dom and G_j reach the boundary condition BC_i in the k-th step, for the first time. Then, the quotient *(D)/(C)* characterizes the *likelihood* of reaching the boundary condition BC_i, exactly in k steps, without violating goal G_j. Intuitively, the smaller this number, the worse, since having a very small value for *(D)/(C)* would mean that when the boundary condition BC_i is reached, there are higher chances of violating goal G_j (i.e., BC_i has severe consequences on the satisfaction of goal G_j).

Again, if we perform the above model counting tasks for increasingly larger values of k, for each of the identified boundary conditions and every goal, we can produce a progression, to facilitate the analysis of how the satisfaction of the goals is affected by the boundary conditions. This information can help the engineer in focusing on those goals that result to be affected to a greater extent, thus suggesting which goals should receive more attention for refinements.

5 EVALUATION

In this section we evaluate our proposal, on various demonstrating examples from the literature on formal requirements specifications. We compute goal-conflicts for each of them, following the approach presented in [15], and then perform model counting experiments as described in the previous section. All the experiments were run on an Intel Core i5 4460 processor, 3.2Ghz, with 8Gb of RAM, running GNU/Linux (Ubuntu 16.04). The scripts to run the experiments, the specifications for all case studies, and a description of how to reproduce the experiments, can be found in **https://dc.exa.unrc.edu.ar/staff/rdegiovanni/ICSE2018.html**.

5.1 Case Studies and Conflict Detection

We evaluate our approach on the following demonstrating examples taken from the literature: the Mine Pump Controller [30], the ATM [39], the London Ambulance Service (LAS) [21], the Rail Road Crossing System [8], the TCP network protocol, and Telephone [18]. Table 3 summarizes the size of each specification, in terms of the corresponding number of domain properties and goals, the number of computed boundary conditions, and the time it took the tool for the corresponding computation. Notice that only for three case studies, namely, MPC, ATM and TCP, the tool produced more than one boundary condition.

Table 3: Case Studies: Computation of Boundary Conditions.

Case Study	Spec. Size	BCs	Time (sec.)
MPC	3	2	11
ATM	3	3	2.71
LAS	5	1	11.68
RRCS	4	1	0.50
TCP	2	2	1.31
Telephone	5	1	11.43

5.2 Goal-Conflict Likelihood Evaluation

We already presented in Table 1 the model counting based analysis of conflicts for the Mine Pump Controller. Table 4 summarizes the results of the same assessment, for the boundary conditions identified for each of the remaining case studies. This information is also graphically depicted, as graphs plotting the number of models of boundary conditions in relation to the models of the corresponding domain properties, as the length trace is increased, in Figure 3.

Notice that, normally, one would expect that models of domain properties grow exponentially as the trace length is increased. A relatively unlikely boundary condition may cover an important number of models for a small k (where the overall number of models for the domain properties is small), but as the trace length increases, the ratio between models of the boundary condition and models of the domain properties should exponentially decrease.

Let us focus, for instance, on the plot for the MPC case study. Notice that the above observation applies to BC_1: as trace length is increased, the probability of reaching BC_1 (i.e., $\Diamond(hw \wedge m)$) decreases exponentially. On the other hand, the probability of reaching BC_2 remains more stable, and in fact quickly becomes more likely than BC_1 as trace length grows. The engineer should prioritize the treatment of BC_2 over BC_1 for resolution.

A similar observation applies to the ATM case study. Observing the ATM boundary conditions plot, we notice that, although initially the probability of reaching BC_1 is much greater than that of reaching BC_2 and BC_3, both BC_1 and BC_2 show an exponential decrease as trace length grows (in fact, BC_2 shows a very small probability of occurring right from the beginning). On the other hand, notice how the probability of reaching BC_3 shows more stability than the others' as trace length is increased. Again, the engineer should prioritize dealing with BC_3 over BC_1 and BC_2.

Regarding the TCP case study, notice that both probabilities remain stable as trace length grows, but BC_1 is more likely than BC_2 while the trace length is increased. The engineer should prioritize dealing with BC_1 over BC_2.

For the other three case studies we have only one identified boundary condition, so no prioritization is needed. Both RRCS and LAS show an exponential decrease in the probability of reaching the corresponding boundary condition as the trace length is increased. In the case of Telephone, on the other hand, the probability shows more stability, again calling for attention.

Table 4: Counting k-bounded word bases for the boundary conditions, for each case study.

Case Study	k												
ATM	1	2	3	4	5	6	10	20	50	75	100	1000	Time
$\#(Dom, k)$	4	32	256	1536	10240	61440	97517568	1.20E+16	2.58E+40	4.87E+60	9.21E+80	8.07E+810	1s
$\#(\{Dom, BC_1\}, k)$	2	10	50	190	830	3226	1023346	2.48E+12	4.12E+31	4.30E+47	4.47E+63	1.94E+640	1s
$\%BC_1$	0.5	0.312	0.195	0.123	0.081	0.052	0.010	0,002	0	0	0	0	-
$\#(\{Dom, BC_2\}, k)$	0	0	2	8	64	256	425984	5.48E+13	1.17E+38	2.22E+58	4.19E+78	3.68E+808	1s
$\%BC_2$	0	0	0.008	0.005	0.006	0.004	0.004	0.004	0.004	0.004	0.004	0.004	-
$\#(\{Dom, BC_3\}, k)$	0	2	16	64	512	2560	4456448	5.74E+14	1.23E+39	2.32E+59	4.39E+79	3.85E+809	1s
$\%BC_3$	0	0.062	0.062	0.042	0.05	0.042	0.046	0.048	0.048	0.048	0.048	0.048	-
TCP	1	2	3	4	5	6	10	20	50	75	100	1000	Time
$\#(Dom, k)$	9	64	512	4096	32768	262144	1073741824	1.15E+18	1.42E+45	5.39E+67	2.03E+90	1.23E+903	1s
$\#(\{Dom, BC_1\}, k)$	3	24	192	1536	12288	98304	402653184	4.32E+17	5.35E+44	2.02E+67	7.63E+89	4.61E+902	1s
$\%BC_1$	0.333	0.375	0.375	0.375	0.375	0.375	0.375	0.375	0.375	0.375	0.375	0.375	-
$\#(\{Dom, BC_2\}, k)$	2	16	128	1024	8192	65536	268435456	2.88E+17	3.56E+44	1.34E+67	5.09E+89	3.07E+902	1s
$\%BC_2$	0.222	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	-
LAS	1	2	3	4	5	6	10	20	50	75	100	1000	Time
$\#(Dom, k)$	129	16384	2097152	2.68E+8	3.43E+10	4.39E+12	1.18E+21	1.39E+42	2.29E+105	1.09E+148	5.26E+210	1.62E+2107	1s
$\#(\{Dom, BC_1\}, k)$	90	3420	129960	4.93E*6	1.87E+8	7.13E+9	1.48E+16	9.33E+31	2.31E+79	7.21E+118	2.25E+158	1.43E+1580	1s
$\%BC_1$	0.697	0.208	0.062	0.018	0.005	0.001	0	0	0	0	0	0	-
RRCS	1	2	3	4	5	6	10	20	50	75	100	1000	Time
$\#(Dom, k)$	32	320	3052	29196	278264	2647284	2.23E+10	1.51E+20	4.76E+49	1.83E+74	7.08E+98	9.10E+983	1s
$\#(\{Dom, BC_1\}, k)$	14	42	258	1686	11514	79026	2.10E+8	7.93E+16	3.85E+42	9.78E+63	2.48E+85	9.05E+855	1s
$\%BC_1$	0.437	0.131	0.084	0.057	0.041	0.0293	0.009	0.005	0	0	0	0	-
Telephone	1	2	3	4	5	6	10	20	50	75	100	1000	Time
$\#(Dom, k)$	9	64	512	4096	32768	262144	1.07E+9	1.15E+18	1.42E+45	5.39E+67	2.03E+90	1.23E+903	1s
$\#(\{Dom, BC_1\}, k)$	0	3	24	192	1536	12288	5.03E+7	5.40E+16	6.69E+43	2.52E+66	9.54E+88	5.76E+901	1s
$\%BC_1$	0.0	0.047	0.047	0.047	0.047	0.047	0.047	0.047	0.047	0.047	0.047	0.047	-

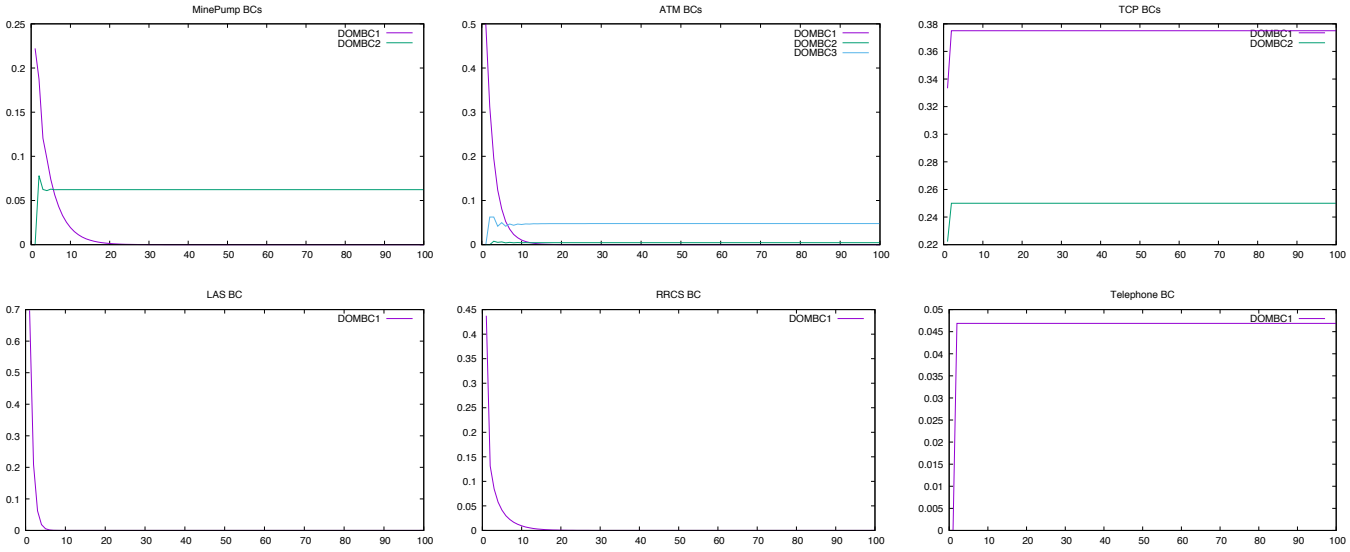

Figure 3: Plotting boundary condition models in relation to domain models, as trace length increases.

Let us now turn our attention to how the identified boundary conditions affect the goals. The relationship between the number of models of each goal, and the number of these models that reach the corresponding boundary condition is graphically plotted in Figure 4 (due to space restrictions, the table is available in the site).

Let us first analyze the MPC case. As it can be seen in the plots, both BCs seriously affects the satisfaction of the goals (recall that, for the effect of boundary conditions on the satisfaction of the goals, the smaller the number, the worse). In particular, notice that boundary condition BC_2, the most likely conflict, affects to a greater extent the satisfaction of goal G_1, compared to its effect on the satisfaction of G_2. So, we would recommend the engineer that goal G_2 ($G(hw \rightarrow \bigcirc(po))$) is the one that should receive more attention, in relation to boundary condition BC_1 ($\Diamond(hw \wedge m)$). A common

mechanism for requirements improvement, in these situations, is goal refinement by strengthening. In fact, the identified situation leads to a well known action for this case study, present in the literature, namely, solving the conflict by strengthening goal G_2 to get rid of the conflict. A solution is based on replacing the original goal with: $G_2' = G(hw \wedge \neg m \rightarrow \bigcirc(po))$.

Regarding the ATM case study, we previously indicated that BC_3 is more likely than the other boundary conditions. In addition, given the plots in Figure 4, we can notice that boundary condition BC_3 affects more the satisfaction of goal G_1 than the satisfaction of goal G_2. So then our approach would suggest the engineer to focus on G_1, in relation to conflict BC_3, for refinements (a closer look at the conflicting situation shows that this boundary condition is enabled by a weak characterization of the domain in relation to account unblocking, that directly affects G_1).

With respect to the TCP example, we previously mentioned that BC_1 is more likely than BC_2. The plots in Figure 4 indicate that BC_1 affects more the satisfaction of goal G_1, while BC_2 strongly affects more the satisfaction of goal G_2. Thus, our approach would recommend the engineer to focus on goal G_1 when resolving conflict BC_1, and focus on goal G_2 when resolving conflict BC_2.

Now, regarding the RRCS case study, notice that $BC1G2$ is constantly zero, indicating that, through boundary condition BC_1, goal G_1 cannot be violated without violating G_2 at the same time. Thus, concentrating in resolving the conflict between BC_1 and G_2, i.e., trying to avoid violating G_2, directly implies that the chances of violating G_1 are decreased.

In the case of Telephone, both goals are equally affected by boundary condition BC_1. In the case of LAS, boundary condition BC_1 affects to a greater extent goals G_2 and G_3, but there is no a significant difference. All the goals are affected in a similar way, suggesting that these should be resolved as a whole.

Regarding the scalability of our approach, notice that the experiments use goal models for which conflicts are computed automatically, and then use model counting on the obtained boundary conditions. While model counting is efficient (the experiments scale well on formula behaviours of up to 1000 steps or more), the complete approach has a bottleneck in the conflict detection phase: it is very costly, due to it being based on LTL tableau [15]. However, if goals and conflicts are provided (e.g., computed previously, manually identified), then the model counting phase can be applied with improved scalability compared to the complete approach. Dealing with more complex case studies requires formal goal models with pre-identified conflicts (most in the literature identify obstacles that affect only one goal, deeming the analysis of conflict impact on different goals less interesting).

Notice that, instead of using our approach, one might think of using [11] with a uniform probabilistic distribution to estimate conflicts likelihood. However, this is different to our approach. A uniform distribution with [11] would apply to all executions of the system, but our approach calculates likelihood using partial executions. E.g., in a goal decomposition into two leaves, imposing a uniform probability implies assigning 50% chances each leaf, overruling other possible executions. In our approach, we use partial executions to analyze in which of these we reach a BC, but other possible partial executions besides the two cases are also considered. See the site for further details.

6 RELATED WORK

Detecting inconsistencies in requirements specifications is a very challenging problem that has received significant attention [24, 25, 29]. Inconsistency management, i.e., how to deal with inconsistencies in requirements, has also been the focus of several studies, in particular on the formal side, e.g., [16, 17, 23, 36, 41]. Much work has been done on the qualitative end, e.g., [22, 35], where the general focus has been on identifying contradictory low-level requirements and computing the degree to which goals are satisfied or denied by them. In general, these approaches focus on the relation between non-functional and behavioural requirements. Our approach tackles a problem of a finer granularity. We propose the use of an approach for identifying inconsistencies, that are captured via the generation of boundary conditions (i.e., declarative expressions) that characterize different conflicting situations, and introduce a novel approach to classify these according their likelihood and severity on the goals. This information can then be used to guide the engineer when goal refinements are required.

In the context of goal-oriented requirements engineering, most of the work contributing to risk analysis [4, 11–13, 43] has been restricted to *obstacles*, a particular kind of conflicts, making them ineffective in situations that arise when the goals themselves are conflicting. The works in [15, 41], on the other hand, focus on goal-conflicts identification, but do not provide any mechanism to assess the criticality of the computed conflicts. In contrast to these works, our approach provides a more general support to risk analysis: it builds upon a mechanism to compute general goals conflicts, not only obstacles, and puts forward a quantitative approach to assess how likely and severe the identified conflicts are.

A work particularly close to our approach is that presented in [11], where a probabilistic framework to propagate obstacle probabilities into the obstacle/goal model is proposed, that enables one to calculate the loss of satisfaction of the obstructed goals. This approach assumes that certain probabilistic information on the domain is provided. More precisely, it assumes that the likelihood of the leaf obstacle is known, having been previously elicited from some experience users, statistical data or some other source. As we mentioned earlier in the paper, our approach tries to help in situation where such information is unavailable, and adopts a model counting mechanism to assess the likelihood of a given identified conflict being reached, and its impact on the goals.

Our work applies to specifications formally captured as LTL formulas. One natural choice would be to resort to an LTL model counter, as that presented in [20], for the purposes of this paper. While the work in [20] deals with the kind of canonical LTL models we are interested in, namely, word models, the approach is only able to deal with safety properties, making it unsuitable for our purposes. Another alternative is to translate the LTL formula to a propositional formula, given a bound k, and then exploiting some propositional model counter to, indirectly, count the number of instances of the original LTL formula. Despite the fact that there exist various efficient propositional model counters, e.g., `RelSAT` [28], `cachet` [37] and `sharpSAT` [38], the reduction to propositional counting leads us to constraints with thousands of variables that cannot be efficiently handled. For these reasons, in this work we use a recently presented string model counter `ABC` [5], that has been demonstrated

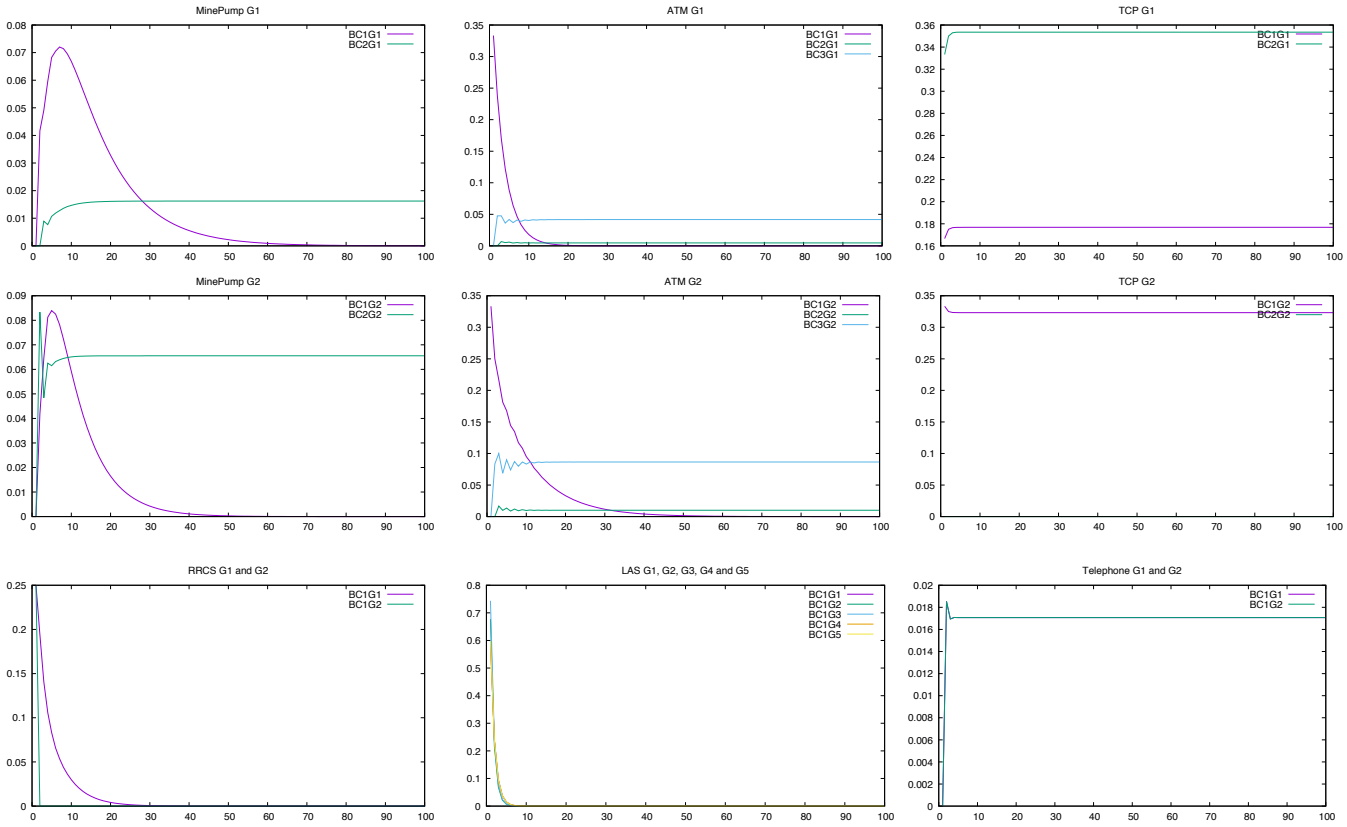

Figure 4: The effect of Boundary Conditions on Goal Satisfaction.

to be very efficient when applied to complex string constraints. The scalability of ABC is related to the fact that it asks for satisfiability only once, producing a *generating function* that can be evaluated for big values of k very efficiently.

Model counting techniques have been recently used in novel ways to provide quantitative information related to relevant software engineering problems. For instance, in [19] a model counting algorithm is defined with the aim of counting the number of data structures that satisfy a given invariant, generalizing model counting techniques that are restricted to linear constraints, and enabling applications in probabilistic software analysis. In [44], a model counter for linear constraints is used in the context of mutation testing, to determine how difficult it is to kill a particular mutant. We consider that our approach also presents an original application of model counting techniques, in our case, to aid in the construction of requirements specifications.

7 CONCLUSION

Getting a correct and sufficiently complete understanding of what a software system to be developed should do is a crucial step toward a successful development project, and the subject of challenging research in software engineering. Among the many problems that arise while producing a software requirements specification, identifying and dealing with inconsistencies in requirements, as early as possible, has a major significance, both from a economical perspective, helping to avoid costly software reworks, and of course in terms of its impact in software quality. In this paper we have presented an approach that builds upon a technique for identifying subtle conflicting situations in requirements specifications, the so called goal conflicts, and proposes the use of modern model counting techniques to assess the criticality of these conflicts, and the severity of their impact in system goals' satisfaction. While related techniques deal with this issue, they require the provision of probabilistic information regarding the likelihood of some system events, a knowledge that must be gathered from statistical data, stakeholders' experience or system simulations. Our approach, on the other hand, is to the best of our knowledge the only one that is able to quantitatively assess goal conflicts and their impact when such information is unavailable. Indeed, our approach does not require a probabilistic model of the environment; instead, it computes probabilities by counting how many models satisfy a conflicting situation, among the models that satisfy the requirements assumptions, and similar kinds of calculations. We showed, through the analysis of various case studies, that this information can be very useful for the engineer, in tasks such as prioritizing the resolution of certain goal conflicts, and directing attention to most affected goals for their refinement.

REFERENCES

[1] Jflap. urlhttp://www.jflap.org.

[2] Lamaconv—logics and automata converter library. urlhttp://www.isp.uni-luebeck.de/lamaconv.

[3] Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[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] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 255–272, 2015.

[6] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, May 2008.

[7] Benjamin Barre, Mathieu Klein, Maxime Soucy-Boivin, Pierre-Antoine Ollivier, and Sylvain Hallé. Mapreduce for parallel trace validation of LTL properties. In Shaz Qadeer and Serdar Tasiran, editors, *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, volume 7687 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2012.

[8] 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.

[9] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[10] Nikolaj Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner, Zohar Manna, Henny Sipma, and Tomás E. Uribe. Verifying temporal properties of reactive systems: A step tutorial. *Formal Methods in System Design*, 16(3):227–270, 2000.

[11] 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.

[12] 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.

[13] 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.

[14] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. In *SCIENCE OF COMPUTER PROGRAMMING*, pages 3–50, 1993.

[15] 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.

[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] Antonio Filieri, Marcelo F. Frias, Corina S. Pasareanu, and Willem Visser. Model counting for complex data structures. In *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*, pages 222–241, 2015.

[20] Bernd Finkbeiner and Hazem Torfah. Counting models of linear-time temporal logic. In Adrian Horia Dediu, Carlos Martín-Vide, José Luis Sierra-Rodríguez, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings*, volume 8370 of *Lecture Notes in Computer Science*, pages 360–371. Springer, 2014.

[21] 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.

[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] 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.

[24] 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.

[25] 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.

[26] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[27] IEEE. Ieee recommended practice for software requirements specifications, 1998.

[28] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island.*, pages 203–208, 1997.

[29] M. Kamalrudin. Automated software tool support for checking the inconsistency of requirements. In *ASE*, pages 693–697, 2009.

[30] J. Kramer, J. Magee, and M. Sloman. CONIC: An integrated approach to distributed computer control systems. In *IEE Proc., Part E 130*, pages 1–10, 1983.

[31] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[32] Timo Latvala, Armin Biere, Keijo Heljanko, and Tommi A. Junttila. Simple bounded LTL model checking. In *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, pages 186–200, 2004.

[33] Jeff Magee and Jeff Kramer. *Concurrency - state models and Java programs (2. ed.)*. Wiley, 2006.

[34] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[35] 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.

[36] 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.

[37] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004.

[38] Marc Thurley. sharpsat - counting models with advanced component caching and implicit BCP. In *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, pages 424–429, 2006.

[39] Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of behavioral models from scenarios. *IEEE Trans. Software Eng.*, 29(2):99–115, 2003.

[40] Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[41] Axel van Lamsweerde, Robert Darimont, and Emmanuel Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. Software Eng.*, 24(11):908–926, 1998.

[42] 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.

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

[44] Willem Visser. What makes killing a mutant hard. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 39–44, 2016.

[45] Eric S. K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, RE '97, pages 226–, Washington, DC, USA, 1997. IEEE Computer Society.