Automated Repair of Unrealisable LTL Specifications Guided by Model Counting

Matías Brizzio
IMDEA Software Institute
Universidad Politécnica de Madrid
Spain

Maxime Cordy
SnT, University of Luxembourg
Luxembourg

Mike Papadakis
SnT, University of Luxembourg
Luxembourg

César Sánchez
IMDEA Software Institute
Spain

Nazareno Aguirre
Universidad Nacional de Río Cuarto
CONICET
Argentina

Renzo Degiovanni
SnT, University of Luxembourg
Luxembourg

ABSTRACT

The reactive synthesis problem consists of automatically producing correct-by-construction operational models of systems from high-level formal specifications of their behaviours. However, specifications are often unrealisable, meaning that no system can be synthesised from the specification. To deal with this problem, we present AuRUS, a *search-based* approach to repair unrealisable Linear-Time Temporal Logic (LTL) specifications. AuRUS aims at generating solutions that are similar to the original specifications by using the notions of *syntactic* and *semantic similarities*. Intuitively, the syntactic similarity measures the text similarity between the specifications, while the semantic similarity measures the number of behaviours preserved/removed by the candidate repair. We propose a new heuristic based on *model counting* to approximate semantic similarity. We empirically assess AuRUS on many unrealisable specifications taken from different benchmarks and show that it can successfully repair all of them. Also, compared to related techniques, AuRUS can produce many *unique* solutions while showing more scalability.

CCS CONCEPTS

• **Software and its engineering → Requirements analysis**; **Formal methods**; **Search-based software engineering**.

KEYWORDS

Search-based Software Engineering, Model Counting, LTL-Synthesis

ACM Reference Format:
Matías Brizzio, Maxime Cordy, Mike Papadakis, César Sánchez, Nazareno Aguirre, and Renzo Degiovanni. 2023. Automated Repair of Unrealisable LTL Specifications Guided by Model Counting. In *Genetic and Evolutionary Computation Conference (GECCO '23), July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3583131.3590454

1 INTRODUCTION

Reactive synthesis is the problem of automatically generating a correct-by-construction implementation for a reactive system, from a given specification of the expected behaviour [7, 9, 12, 24, 40, 46]. The specification comprises the list of the variables controlled by the environment and system (i.e., inputs and outputs), respectively, and a declarative description of the expected properties of the system, i.e., the *goals*. Based on these, reactive synthesis produces a model for the system, usually referred to as *controller*, which interacts with the environment, and guarantees the specified goals [46].

Temporal logic is widely considered the standard formalism for expressing the expected goals of reactive systems [38]. Reactive specifications are usually expressed as *assume-guarantee assertions*, i.e., $A \Rightarrow G$, where A captures *assumptions* on the behaviour of the environment, and G expresses the goals the system must fulfill, provided that the assumptions are met. Since the specifications are the central element in the synthesis, their quality is crucial for a successful process.

Reactive specifications often contain imperfections that make them *unrealisable*, i.e., no controller can be synthesised. Common issues leading to unrealisability are (1) inconsistencies between goals making them unsatisfiable (and therefore unrealisable); (2) inadequate assumptions and guarantees, which allow the environment to satisfy the assumptions and prevent the system from complying with the guarantees. Therefore, arriving at a consistent *and* realisable specification is not straightforward and demands comprehensive elicitation activities to prevent these common issues.

Previous attempts have made significant efforts to provide automated mechanisms to assist engineers in identifying and resolving sources of unrealisability in temporal logic specifications. Some concentrate on *diagnosing* the cause of synthesis impossibility, e.g., by computing a core of assertions that make the specification unrealisable [16, 29, 48]. Some generate counter-strategies that evidence how the environment prevents the controller from satisfying the goals [47]. Related to our work, many approaches to repair unrealisable specifications have been presented [8, 13, 14, 37, 41]. These techniques attempt to repair the specifications *just* by adding assumptions that are built from the information extracted from the generated counter-strategies, and the current specification. This is a significant limitation because they do not consider that unrealisability might be caused by failures in the current assumptions and guarantees. These techniques also impose syntactical restrictions on how specifications are written, limiting their application to particular patterns (e.g., GR(1), a subset of LTL).

In this paper, we present AuRUS, a *search-based* approach to repair unrealisable specifications, which applies to LTL [38] specifications and generates candidate repairs by changing both, assumptions and guarantees. AuRUS consists of a genetic algorithm (GA) that, given an unrealisable LTL specification, attempts to generate a *realisable* variant of it, which is as close as possible to the original one. The algorithm iteratively explores candidate repairs of the original specification, seeking to find a realisable variant that is both syntactically and semantically similar to the original one. The syntactic similarity is measured by using the number of *sub-formulas* that belong to both the original and the mutated specification. On the other hand, semantic similarity is measured by calculating the number of behaviours from the original specification that were maintained in the candidate repair. These behaviours are computed by using *model counting*. Since all existing model counting approaches for LTL do not scale well, we develop an alternative approach to approximate the LTL model counting problem which considerably improves scalability. We empirically assess AuRUS and show that it is effective at repairing unrealisable specifications not handled by previous techniques, while also producing more *unique* repairs not computed by related approaches.

2 PRELIMINARIES

2.1 Linear-Time Temporal Logic (LTL)

LTL is a logical formalism widely employed to specify reactive systems [38]. Given a set AP of propositional variables, LTL formulas are inductively defined using the standard logical connectives, and the temporal operators $\bigcirc$ (next) and $\mathcal{U}$ (until), as follows: *(i)* every $p \in AP$ is an LTL formula, and *(ii)* if φ and ψ are LTL formulas, then so are $\neg\varphi$, $\varphi \vee \psi$, $\bigcirc\varphi$ and $\varphi\mathcal{U}\psi$. Other connectives and operators, such as $\wedge$, $\Box$ (always), $\Diamond$ (eventually), and $\mathcal{W}$ (weak-until), can be defined in terms of the basic ones. LTL formulas are interpreted over infinite traces of the form $\sigma = s_0\ s_1 \ldots$, where each s_i is a propositional valuation on 2^{AP}. Formulas with no temporal operator are evaluated in the first state of the trace. Formula $\bigcirc\varphi$ is true in σ iff φ is true in $\sigma[1..]$, i.e., the trace obtained by removing the first state from σ. Formula $\varphi\mathcal{U}\psi$ is true in σ iff there exists a position i in the trace, such that, ψ is true in $\sigma[i..]$ and for every $0 \leq k < i$, φ is true in $\sigma[k..]$. An LTL formula φ is *satisfiable* (SAT) iff at least one trace satisfies φ. The *model counting* problem consists of calculating the number of models satisfying φ. In the case of LTL, if a formula is unsatisfiable, the number of models is zero. Otherwise, it has an infinite number of models. Therefore, LTL model-counting is restricted to *bounded models* [25], i.e., it computes how many models up to k states exist for φ and a bound k. AuRUS uses bounded model counting to guide the search by measuring the semantic impact of syntactic changes on candidate repairs, in terms of the number of preserved and removed behaviors from the original specification.

2.2 Reactive LTL Synthesis

Reactive LTL synthesis is the problem of automatically constructing a reactive module that reacts to the environment with the objective of realizing a given LTL specification of the form assume-guarantee, $\varphi : A \rightarrow G$ [46], defined over a set of variables $\mathcal{V} = \mathcal{X} \cup \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are the variables controlled by the environment, and system, respectively. A strategy for φ is a function $\sigma : (2^{\mathcal{X}})^{+} \rightarrow 2^{\mathcal{Y}}$ that maps finite sequences of subsets of $\mathcal{X}$ into subsets of $\mathcal{Y}$. For an infinite sequence $X = X_1, X_2, \ldots \in (2^{\mathcal{X}})^{\omega}$, the play induced by strategy σ is the infinite sequence $\rho_{\sigma,X} = (X_1 \cup \sigma(X_1))(X_2 \cup \sigma(X_2)))\ldots)$. A play ρ is *winning* if $\rho \models \varphi$. A strategy is winning when $\rho_{\sigma,X} \models \varphi$ for all $X \in (2^{\mathcal{X}})^{\omega}$.

Realisability is the problem of deciding whether a specification has a winning strategy, and synthesis is the problem of computing one. The Unrealisability of a specification means that no winning strategy exists for the system. This implies that the environment can always falsify the specification, no matter which strategy the system chooses. AuRUS performs syntactic changes in the specification to remove the flaw that makes it unrealisable. AuRUS delegates the realisability check to Strix [43], one of the most efficient synthesis tools presented at the annual synthesis competition [3].

2.3 Genetic Algorithms

Genetic algorithms [27, 30, 44] are heuristic search algorithms, inspired by natural evolution. Candidate solutions are called *individuals* or *chromosomes*, and are often represented as sequences of *genes* (characteristics) that capture their features. Genetic algorithms maintain a *population* of candidate solutions, rather than a single "current" candidate, as in traditional search. They are largely driven by random decisions, e.g., in the generation of the initial population, and how the new candidate solutions are generated from existing ones. To produce new individuals, it exploits information in the current population, combining their characteristics (called *crossover*), or randomly altering the information in specific individuals (called *mutation*). The effectiveness of this general search process is guided by a heuristic function, called *fitness function*. Intuitively, this function measures how "fit" a particular individual is, i.e., how close it is to being a real solution to the search problem under consideration. This evolution process is usually performed until some termination criterion is met, e.g., a defined number of iterations (known as *generations* of the population). AuRUS employs genetic algorithms to search for realisable repairs, close to the unrealisable specification given as input. Individuals in our case will represent LTL specifications, the genetic operators will produce new specifications from others, and the fitness function will attempt to evaluate how "close" a candidate repair is to be realisable, as well as how close is to the original (unrealisable) one.

3 A MOTIVATING EXAMPLE

Let us present a running example to illustrate the main ideas behind AuRUS. Consider the problem of synchronising the access to a shared resource, via an arbiter [31]. Two processes request access to the resource via signals r_1 and r_2, respectively. An extra signal a indicates when the resource can be accessed. The arbiter indicates which process has been granted access by means of respective signals g_1 and g_2. Signals $r1$, $r2$ and a thus constitute the *inputs*, while signals $g1$ and $g2$ are the *outputs*. The following guarantees are elicited in [31] for this problem:

$$\boldsymbol{G_1} : \Box(r_1 \rightarrow \Diamond g_1) \quad \boldsymbol{G_2} : \Box(r_2 \rightarrow \Diamond g_2) \quad \boldsymbol{G_3} : \Box(\neg a \rightarrow (\neg g_1 \wedge \neg g_2))$$

Intuitively, G_1 states that if the first process requests access to the resource, the arbiter will eventually grant it. Guarantee G_2 states the same but for the second process. While guarantee G_3 states that if the resource cannot be accessed, no process is granted permission. No assumptions were identified for this specification. Hence, the specification $S = G_1 \wedge G_2 \wedge G_3$ is *unrealisable*.

A cause of unrealisability is that the environment is allowed to set the input signal a to *false* continuously, preventing the arbiter from granting access to the resource (see G_3). Therefore, if any of the processes requests access to the resource in such a situation, no implementation would satisfy the guarantees G_1 and G_2. In [31], authors propose different alternatives for "fixing" these issues. One option is to add an environment assumption to ensure that the resource is allowed to be accessed infinitely often; indeed, by adding the assumption $A_1 = \Box\Diamond a$, the resulting specification becomes realisable. Another option explored in [31] is to indicate that the arbiter will enforce mutual exclusion in accessing the resource. This is done by replacing G_3 by an alternative guarantee $G_3' = \Box(\neg(g_1 \wedge g_2))$. The resulting specification $S' = G_1 \wedge G_2 \wedge G_3'$ is also realisable. The overall intuition we get from this example is that, in cases of unrealisability, it can sometimes be fixed by making *small* changes to the original specification. But, even when the repairs were generated automatically, a domain expert would need to review them, and decide which of them is an acceptable solution. For instance, the repairs introducing a new assumption would need to be analysed to check if such an assumption is reasonable to expect from the environment. Thus, to help the domain expert and make easier this validation activity, we would like to maintain as much as possible from the original specification in the generated repairs, and simply modify what is necessary to make it realisable. In this paper, we propose AuRUS, a genetic algorithm that performs syntactic modifications to an unrealisable specification, with the aim of producing a set of realisable candidate repairs. The candidate repairs are searched for in the *"vicinity"* of the original specification, in the sense that they aim at being slight syntactic and especially semantic, modifications of the original one. AuRUS has the challenge of dealing with a very large search space of LTL specifications that are obtained by performing syntactic changes to the original one. It also needs to objectively quantify the semantic impact of each change. AuRUS is guided by a multi-objective fitness function that attempts to minimise syntactic and semantic changes, while at the same time attempting to achieve realisability. Let us provide some intuition on how AuRUS works, and how it can produce some solutions presented in [31].

AuRUS starts by generating an initial population that represents samples of candidate solutions. These are generated by introducing new assumptions based on patterns commonly found in reactivity specifications [23]. For example, assumptions stating that input events occur infinitely often ($\Box\Diamond r_1$, $\Box\Diamond r_2$, $\Box\Diamond a$), and that different input events cannot simultaneously occur ($\Box\neg(r_1 \wedge r_2 \wedge a)$), will be considered. In our running example, the initial population already contains one of the realisable solutions proposed in [31]. Other candidate solutions can be obtained by the successive application of genetic operators to some selected specifications. AuRUS implements the two most common genetic operators, *crossover* and *mutation.* Given two specifications S_1 and S_2, the crossover operator will produce a new specification S_3 by replacing a sub-formula of S_1, by a sub-formula of S_2. For instance, if both S_1 and S_2 are exactly the same specification $G_1 \wedge G_2 \wedge G_3$, the crossover operator can produce a new specification $S_3 = G_1 \wedge G_2' \wedge G_3$, where $G_2' = \Box(r_2 \rightarrow \Diamond g_1)$ is obtained by replacing sub-formula g_2 in G_2, by the sub-formula g_1 extracted from G_1 in S_2. On the other hand, the mutation operator will create a new specification by applying a syntactic mutation to some sub-formula of the specification. For instance, mutating generated specification S_3, the algorithm can produce a new specification $S_4 = G_1' \wedge G_2' \wedge G_3$ in which $G_1' = \Box(r_1 \rightarrow \bigcirc g_1)$ is obtained by changing the operator $\Diamond$ by $\bigcirc$ in G_1. Going back to the arbiter solutions, to obtain the realisable version $G_1 \wedge G_2 \wedge G_3'$, two mutations to the original guarantee G_3 are necessary: first, a replacement of the sub-formula a by *false* leading to $\Box(\neg g_1 \wedge \neg g_2)$; and then the replacement of operator $\wedge$ by $\vee$, obtaining the formula $\Box(\neg g_1 \vee \neg g_2)$ which is equivalent to G_3'. AuRUS may produce other realisable specifications that can also be considered as candidate repairs. The fitness function plays a *crucial* role in guiding the search. Intuitively, the *fitness function* is the oracle that is used to assess the quality of the candidate solutions, giving higher scores to "better" individuals, i.e., those closer to sought solutions. AuRUS implements a multi-objective fitness function that assesses three key properties of the candidate solutions: *(1)* it checks whether the specification is realisable or not; *(2)* it then computes the syntactic similarity with respect to the original specification; and finally *(3)* it computes the semantic similarity with respect to the original specification (i.e., the number of behaviours preserved and removed by the formula modifications). We show in Section 6 the importance of each fitness factor to guide the search toward adequate solutions, i.e., solutions that are very alike to the original.

4 AURUS APPROACH

AuRUS takes as input an unrealisable specification S, and by the successive application of genetic operations, it aims at producing a specification S' that is realisable, and minimizes the syntactic and semantic changes with respect to S.

4.1 Search Space and Initial Population

Individuals in our search space are LTL specifications $S = (A, G)$ over $\mathcal{V}$, consisting of assumptions and guarantees. Genetic operators are used to produce new individuals with syntactic changes to both assumptions and guarantees, with equal probability. AuRUS begins by creating assumptions based on patterns commonly found in reactive specifications [23] to form the initial population. These new assumptions are generated from the original specification S, resulting in $S_0 = (A \cup a_0, G)$, where a_0 follows the patterns: (1) $\Box\Diamond x_i$, (2)$\Box\neg(x_0 \wedge \ldots \wedge x_n)$, and (3) $\Box\Diamond(x_0 \wedge \ldots \wedge x_n)$, for $x_i \in \mathcal{X}$. These patterns respectively express that the input x_i holds infinitely many times, that all input events cannot happen at the same time, and that all input events hold at the same time, infinitely many times. We only include input variables in the assumptions to prevent trivial solutions. Adding assumptions like $\Box y$, $y \in \mathcal{Y}$ would result in a specification that can easily be satisfied by setting y to false, making it trivially realisable. AuRUS avoids anomalous solutions [22] in the initial population and checks for unsatisfiable assumptions during the search to prevent trivially realisable repairs.

4.2 Genetic Operators

AuRUS implements the two most common genetic operators such as, *crossover* and *mutation*, adapted to LTL. Let $SF(\varphi)$ be the list of sub-formulas of φ, e.g., $SF(\Box\neg p) = [\Box\neg p, \neg p, p]$. We denote by $\varphi[\phi\backslash\psi]$ the formula that is obtained by replacing occurrences of ϕ in φ, by ψ. For instance, $\Box(\neg p)[\neg p\backslash r]$ returns $\Box(r)$. Given two formulas φ and φ', we define: $replaceSub(\varphi, \varphi') = \varphi[\phi\backslash\psi]$ and $combineSub(\varphi, \varphi') = \varphi[\phi\backslash\phi \bullet \psi]$, s.t. $\phi \in SF(\varphi)$, $\psi \in SF(\varphi')$, and $\bullet \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$. Intuitively, $replaceSub(\varphi, \varphi')$ returns a new formula that consists of replacing a sub-formula of φ with a sub-formula of φ'; while $combineSub(\varphi, \varphi')$ takes a sub-formula from φ and combines it with another from φ', using a binary operator.

The crossover operator combines two LTL specifications, namely $S_1 = (A_1, G_1)$ and $S_2 = (A_2, G_2)$, to create a new specification $S_3 = (A_3, G_3)$. This is achieved by merging the assumptions and guarantees of the original specifications.

Particularly, every assumption in A_3 is taken from A_1 or A_2, or is generated by replacing ($replaceSub(a_1, a_2)$) or combining ($combineSub(a_1, a_2)$) sub-formulas from A_1 and A_2, where $a_i \in A_i$. To produce G_3, the crossover operator performs the same choices but it takes guarantees from G_1 and G_2 instead.

The mutation operator takes a specification and applies syntactic modifications to produce a new one. Given a specification $S = (A, G)$, the algorithm randomly selects one assumption/guarantee to which the mutation will be applied. Let $mutate(\varphi) = \varphi'$ be the function that takes a formula and produces a mutation of it. When the mutation operator is applied to some assumption $a \in A$, it returns a new specification $S' = (A', G)$, where $A' = (A\backslash\{a\}) \cup \{a'\}$ and $a' = mutate(a)$. The same applies when some guarantee $g \in G$ is mutated. *mutate* is defined as follows:

(1) if $\phi = b$ or $\phi = p$, where $b \in \{true, false\}$ and $p \in AP$, then:
  (a) $\phi' = b'$, s.t. $b' \in \{true, false\}$ and $b \neq b'$.
  (b) $\phi' = q$, s.t. $q \in AP$ and $p \neq q$.
  (c) $\phi' = o_1\phi$ where $o_1 \in \{\Box, \Diamond, \bigcirc, \neg\}$.
(2) if $\phi = o_1\phi_1$, where $o_1 \in \{\neg, \bigcirc, \Diamond, \Box\}$, then:
  (a) $\phi' = mutate(\phi_1)$.
  (b) $\phi' = o_1' mutate(\phi_1)$, s.t. $o_1' \in \{\neg, \bigcirc, \Diamond, \Box\}$.
  (c) $\phi' = o_1' o_1 mutate(\phi_1)$ where $o_1' \in \{\neg, \bigcirc, \Diamond, \Box\}$.
  (d) $\phi' = p\ o_2'\ o_1'(mutate(\phi_1))$, s.t. $p \in AP$, $o_2' \in \{\mathcal{U}, \mathcal{W}, \wedge, \vee\}$ and $o_1' \in \{\neg, \bigcirc, \Diamond, \Box\}$.
(3) if $\phi = \phi_1 o_2 \phi_2$, where $o_2 \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$, then:
  (a) $\phi' = mutate(\psi_i)$, s.t. $\psi_i \in \{\phi_1, \phi_2\}$
  (b) $\phi' = mutate(\phi_1)\ o_2'\ mutate(\phi_2)$, s.t. $o_2' \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$.
  (c) $\phi' = o_1'(mutate(\phi_1)\ o_2'\ mutate(\phi_2))$, s.t. $o_1' \in \{\neg, \bigcirc, \Diamond, \Box\}$ and $o_2' \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$.

4.3 Fitness Function

AuRUS is guided by a multi-objective fitness function that aims at generating a realisable variant, as close as possible, to the unrealisable specification given as input. This function focuses on checking three key properties of the candidate solutions, with the objective of finding one that is *realisable*, and that minimises the *syntactic* and *semantic* changes with respect to the original specification. Syntactic similarity is measured in terms of the number of sub-formulas shared by the original specification and the candidate. Semantic similarity is measured in terms of the number of behaviours that were maintained in the candidate. Basically, let S be the unrealisable specification given as input, the fitness value for a candidate repair S' is computed by the following function f:

$$f(S') = \alpha * status(S') + \beta * synSim(S, S') + \gamma * semSim(S, S')$$

where $status(S')$ focuses on checking if S' is satisfiable and realisable; while, $synSim(S, S')$ and $semSim(S, S')$ compute the syntactic and semantic similarities between S and S', respectively. Constants α, β and γ are the factors that assign different weights to the three properties of interest in the candidate repair S'.

Let $S' = (A', G')$ be a candidate solution, we define $status(S')$ as follows:

$$status(S') = \begin{cases} 1 & \text{if } A' \wedge G' \text{ is satisfiable and } A' \to G' \text{ realisable;} \\ 0.5 & \text{if } A' \wedge G' \text{ is satisfiable, but } A' \to G' \text{ unrealisable;} \\ 0.2 & \text{if } A' \wedge G' \text{ is unsatisfiable, but } A'/G' \text{ are satisfiable;} \\ 0.1 & \text{if } A' \text{ is satisfiable, but not } G'\text{;} \\ 0 & \text{if } A' \text{ is unsatisfiable.} \end{cases}$$

$status(S')$ will return 1 iff S' is both, satisfiable and realisable. When the candidate S' is satisfiable, but still unrealisable, $status(S')$ will return 0.5. Whether assumptions/guarantees are unsatisfiable, it will return values closer to 0.

$synSim(S, S')$ computes the *syntactic similarity* between specifications S and S', measured in terms of the number of sub-formulas that belong to both the original S and the candidate S':

$$synSim(S, S') = 0.5 * \left(\frac{\#(SF(S) \cap SF(S'))}{\#SF(S)} + \frac{\#(SF(S) \cap SF(S'))}{\#SF(S')} \right)$$

Small values for $synSim(S, S')$ indicate that S' is syntactically very different from S, while values closer to 1 indicate that both specifications are very similar. The fitness function uses this value to quantify the syntactic changes produced by the genetic operators.

Small changes in the syntax of a specification may result in significant changes in its semantics. For example, a mutation that negates a formula may appear as a single syntactic change, but it can completely reverse the behaviors described by the original formula. The function $semSim(S, S')$ computes the *semantic similarity* between specifications S and S', measured in terms of the number of behaviours from the original specification still in the candidate repair. To automatically compute this value, we rely on LTL *model counting*. Given a specification S and a bound k, let us denote by $\#(S, k)$ the number of models up to k states satisfying S. We define $semSim(S, S')$ as follows:

$$semSim(S, S') = 0.5 * \left(\frac{\#(S \wedge S', k)}{\#(S, k)} + \frac{\#(S \wedge S', k)}{\#(S', k)} \right)$$

Notice that small values for $semSim(S, S')$ indicate that the behaviours described by S are very different from the ones described by S'. In particular, when $S \wedge S'$ is unsatisfiable, $semSim(S, S')$ is 0. As this value gets closer to 1, both specifications characterise an increasingly large number of common behaviours. Since the computation of $semSim(S, S')$ requires calculating several model counting instances, it is crucial for AuRUS to perform model counting efficiently. Thus, later in Section 5, we develop an automata-based estimation for LTL model counting, that scales much better than exact LTL model counting algorithms. The user can select different values for the different parameters that might affect the fitness function, e.g., α, β, γ, and the bound k. We tested AuRUS in various complex specifications, selecting the best-performing configuration and comparing its effectiveness to previous techniques in Section 6.

4.4 Selection

The fittest individuals are chosen for the next generation using the traditional *“best selector”* operator, sorting individuals by fitness and selecting the best until maximum population size.

4.5 Soundness, and (In)completeness

AuRUS guarantees satisfiability and realisability of its generated repairs, checked using Polsat [35] and Strix [43], respectively. While AuRUS uses a non-exhaustive search and may not consider all possible repairs, its genetic operators are complete, allowing the production of φ' from φ via crossover and mutation.

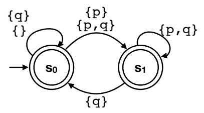
5 LTL MODEL COUNTING APPROXIMATION

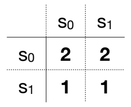
AuRUS proposes to use LTL model counting to compute the semantic similarity $semSim(S, S')$. Unfortunately, exact LTL model counting techniques quickly reach their scalability limits [25]. As a consequence, the work presented in [19] attempts to deal with the scalability issues. It presents a translation from LTL formulas to regular expressions, such that, each string recognized by the regular expression is a prefix of some trace satisfying the formula. Then, a string model counter (ABC [10]) is used to compute the number of strings that, up to a certain length, satisfy the regular expression. Though effective, this approach works on specific examples but fails to scale overall (failing in 22/26 specifications, see Section 6).

Thus, we develop a new technique to efficiently estimate the LTL model counting problem. Intuitively, we aim at counting the number of prefixes satisfying an LTL formula and use that number as a proxy for the number of models of the formula. To improve scalability, we employ matrices multiplication for this task (as ABC [10]).

Basically, our approach operates in two steps. First, we rely on established algorithms to generate an automaton A_φ that recognises all the traces satisfying a given formula φ. Recall that LTL formulas are interpreted on infinite traces (c.f. Section 2.1). This means that every trace satisfying φ, has a path in automaton A_φ, in which some *accepting state* is visited infinitely many times (i.e., a loop). Notice that, every *finite* path reaching some accepting state of A_φ, is potentially a *prefix* of some trace recognised by A_φ. Then, we can have an estimation of the number of traces accepted by A_φ, by counting the number of finite paths of A_φ that reach some accepting state (this can be thought as if we interpreted accepting states as final states). Notice that, it may happen that our approach overestimates the number of models for φ. For instance, when some reachable accepting state does not loop. Conversely, the accepting state may be part of multiple loops; in this case, the prefix accepted by A_φ corresponds to multiple traces, which leads to underestimating the number of models for φ. These are the reasons why our approach only *approximates* the number of models.

Then, we encode automaton A_φ into a $N \times N$ transfer matrix T_φ, where N is the number of states in A_φ, such that the value of each $T_\varphi[i, j]$ denotes the number of transitions from states i to j in A_φ. The number of finite paths, of length k, reaching some accepting state of A_φ, can be computed by solving $I \times T_\varphi^k \times F$, where I is a row vector codifying the initial states; T_φ^k is the matrix resulting from multiplying k times matrix T_φ; and F is a column vector codifying the final (accepting) states of A_φ. For example, Fig. 1 and Fig. 2 show the automaton and the transfer matrix, generated by our approach





	s0	s1
s0	2	2
s1	1	1

Figure 1: Finite automaton. Figure 2: Transfer matrix.

from the formula $\psi = \Box(p \to \bigcirc q)$. Considering $k = 4$, our model counting approach answers that there are 108 models, computed by the following matrices multiplication:

$$I \times T_\varphi^4 \times F = I \times \begin{bmatrix} 2 & 2 \\ 1 & 1 \end{bmatrix}^4 \times F = \begin{bmatrix} 1 & 0 \end{bmatrix} \times \begin{bmatrix} 54 & 54 \\ 27 & 27 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 108$$

Actually, there are exactly 351 lasso traces of length 4 for ψ but our approach reports 108 (i.e., the approximate number of prefixes for these traces). However, we show in Section 6.5 that it provides a good estimation of the exact model counting. Meaning that, if it computes that formula φ has more prefixes of length k than formula ψ, then it is almost sure that the number of lasso traces of φ of length k is greater than the number of lasso traces for ψ.

6 EXPERIMENTAL EVALUATION

We evaluate AuRUS around the following research questions:

RQ1 *How effective and efficient is AuRUS?*
RQ2 *How does it compare with random generation?*
RQ3 *Does AuRUS generate unique solutions?*
RQ4 *How does each objective of the fitness function contribute to AuRUS’ effectiveness?*
RQ5 *What is the precision of our model counting method?*

We answer RQ1-RQ4 using unrealisable specifications from the literature and benchmarks, and RQ5 using randomly picked LTL formulas from an LTL SAT solving benchmark.

Specifications. We consider 26 unrealisable specifications in our evaluation. Table 1 summarises for each specification the number of input/output variables, assumptions (A), and guarantees (G). We consider 5 cases from the literature, 13 from SYNTCOMP [4], and 8 specifications created by students and reported in SYNTECH15 [41].

Table 1: Unrealisable Specifications.

Literature (5)	#In-Out	#A-#G
Arbiter	3-2	0-3
MinePump	2-1	1-2
RG1	2-2	1-4
RG2	2-1	0-2
Lift	3-3	7-12

SYNTECH15 (8)	#In-Out	#A-#G
Humanoid458	3-10	0-11
Humanoid503	6-11	1-17
Humanoid531	1-11	2-17
Humanoid741	4-14	5-21
Humanoid742	1-14	2-26
GyroV1	3-3	6-7
GyroV2	3-3	7-7
PCarV2-888	3-9	4-21

SYNTCOMP (13)	#In-Out	#A-#G
Detector	2-1	0-4
Full Arbiter	3-3	0-16
Lily02	3-1	0-3
Lily11	2-2	0-1
Lily15	2-2	0-5
Lily16	3-3	0-9
Load Balancer	3-2	3-8
ltl2dba_R_2	2-1	0-1
ltl2dba_theta_2	4-1	0-1
ltl2dba27	1-1	0-1
Prioritized Arbiter	4-4	1-10
Round-Robin	2-2	2-4
Simple Arbiter	2-2	0-4

Implementation. AuRUS uses OWL [32] to manipulate LTL specifications. Apache Commons Math [2] to manipulate matrices for model counting. AuRUS also integrates Polsat [35], a portfolio that runs 4 LTL solvers in parallel. Moreover, AuRUS uses Strix [43] to check realisability. The experiments in this section were conducted on a cluster with Xeon 2.6GHz, with 16Gb of RAM, running GNU/Linux. The tool, case studies, and a description of how to reproduce the experiments can be found in the replication package **https://sites.google.com/site/unrealrepair/**.

Experimental Setup. As AuRUS is driven by random decisions, for each experiment, we run it 10 times. Precisely, in our experimentation AuRUS is configured as follows: the population size is 100, the model-counter bound k is 20, best selector, the crossover operator is applied to 10% of the individuals, and the mutation operator is applied to each individual, to which each gene (sub-formula) is mutated with a probability of $1/N$ (where N is the size of the formula). The termination criterion is reached either when 1000 individuals are generated or after 2hrs of execution time.

Notice that, our fitness function (Sec. 4.3) focuses on three aspects of the candidate solution (the status (α), the syntactic (β) and semantic (γ) similarities). We assess the performance of AuRUS under many configurations, by considering values for α, β, and γ, such that $\alpha + \beta + \gamma = 1$ (i.e., the sum of weights is 100%). We organise our unrealisable specifications into two disjoint sets: the *development* set, and the *evaluation* set. We randomly selected 6 cases to be part of the development set (2 from the literature, 2 from SYNTECH15, and 2 from SYNTCOMP) and the remaining 20 cases as part of the evaluation set. The development set is meant to support us in setting the parameters of our algorithm, with the hope that the best-performing configuration for the development set, will generalise to the evaluation set. To find the best-performing configuration, for each experiment we measure the number of repairs generated by AuRUS, as well as, the syntactic and semantic similarities of the found solutions. Then, for each case, we can compare the performance of two configurations by using the Vargha-Delaney A ($\hat{A}_{12}$) measure [53], to determine which configuration obtained better performance for that particular subject. This will allow us to analyse which configuration generalises more and better to all the case studies. Particularly, realisability checking is crucial for quality and quantity of solutions. Also, better performance is reached when the weight assigned to the semantic similarity is greater or equal to the one assigned to the syntactic similarity. For instance, configurations ($\alpha = .7, \beta = .1, \gamma = .2$), ($\alpha = .8, \beta = .07, \gamma = .13$), or ($\alpha = .9, \beta = .05, \gamma = .05$) typically reach better performance.

6.1 Effectiveness and Efficiency Evaluation

Table 2 summarises the average results, out of the 10 runs, obtained with the best-performing configuration for each case study. We report the average number of repairs and time (in seconds) per case required by AuRUS to explore the 1000 individuals. Noticeable, AuRUS succeeds in generating satisfiable and realisable repairs in 100% of the runs. As expected, it required more time to analyse the more complex specifications such as the Lift, full arbiter, and Humanoid cases in which the 2 hours timeout was reached. Particularly, there are four cases for which the algorithm could find just a few repairs: on average, 4 repairs for the full arbiter, 13 for the prioritized arbiter, and 3 repairs for Humanoid503 and PCarV2-888. This is because these cases contain several guarantees that require many changes by the algorithm to finally find realisable solutions. In particular, the first 2 cases are part of the synthesis competition and were made artificially unrealisable (by adding assertions contradicting the existing ones) to use them for assessing the efficiency of the tools participating in the competition.

Table 2: Comparison between AuRUS and random.

Literature	Tech.	#Sol.	Time
Arbiter	AuRUS	467	921
	Random	11	404
Minepump	AuRUS	481	897
	Random	31	678
RG1	AuRUS	380	905
	Random	15	529
RG2	AuRUS	459	935
	Random	27	406
Lift	AuRUS	303	3170
	Random	0	930

SYNTECH15	Tech.	#Sol.	Time
GyroV1	AuRUS	530	1574
	Random	7	724
GyroV2	AuRUS	618	1388
	Random	40	771
Humanoid458	AuRUS	582	2307
	Random	26	738
Humanoid503	AuRUS	3	7400
	Random	1	616
Humanoid531	AuRUS	86	7400
	Random	19	239
Humanoid741	AuRUS	80	7400
	Random	11	756
Humanoid742	AuRUS	99	7400
	Random	21	705
PCarV2-888	AuRUS	3	7400
	Random	4	174

SYNTCOMP	Tech.	#Sol.	Time
Detector	AuRUS	522	1799
	Random	21	1592
Full Arbiter	AuRUS	4	3805
	Random	6	1003
Lily02	AuRUS	387	2656
	Random	4	427
Lily11	AuRUS	623	834
	Random	35	350
Lily15	AuRUS	424	1643
	Random	4	1248
Lily16	AuRUS	385	1756
	Random	6	984
Load Balancer	AuRUS	532	1619
	Random	29	801
ltl2dba_R_2	AuRUS	623	1442
	Random	42	1451
ltl2dba_theta_2	AuRUS	660	1453
	Random	32	1493
ltl2dba27	AuRUS	582	1473
	Random	46	1048
Prioritized Arbiter	AuRUS	13	4205
	Random	24	5680
Round-Robin	AuRUS	678	1713
	Random	76	904
Simple Arbiter	AuRUS	504	1012
	Random	17	404

6.2 Comparison with Random Generation

Random starts by producing 1000 syntactic variants of each unrealisable specification and then checks which one is satisfiable and realisable. The random approach uses the same mutation operator as AuRUS to produce syntactic modifications to the original specifications, with the additional requirement that at least one sub-formula (assumption/guarantee) has been modified. We repeat this experiment 10 times and report all results in Table 2. Notice that, unsurprisingly, the time required by random is considerably smaller than the required by AuRUS in most cases, except for cases ltl2dba_R_2, ltl2dba_theta_2 and Prioritized Arbiter. However, random effectiveness is relatively low compared to AuRUS, producing on average 23 times less repairs than AuRUS. We also compute, for each case study, the $\hat{A}_{12}$ measure to compare the number of repairs obtained in the 10 runs of our best-performing configuration and random. $\hat{A}_{12}$ results to be 100% for almost every case study, indicating that AuRUS obtains more repairs than random in every run. There are only 3 exceptions where random produces more realisable solutions: in Full Arbiter ($\hat{A}_{12}$ of 33.3%), Prioritized Arbiter ($\hat{A}_{12}$ of 10.5%) and PCarV2-888 ($\hat{A}_{12}$ of 33.3%). Fig. 3 shows further details regarding random performance.

6.3 Comparison with Related Approaches

We study to what extent AuRUS's repairs are unique or related to the repairs produced by other approaches. Precisely, we analyse if AuRUS is able to produce some equivalent, weaker, or stronger repair. We say that a formula B is weaker than A, if $A \rightarrow B$ holds (i.e. if $A \wedge \neg B$ is unsatisfiable). Typically, weaker specifications are thought of as more general solutions, while stronger ones correspond to more localised solutions. Our intention is twofold: we want to show that AuRUS can produce repairs that are close to the ones obtained by other approaches, and also that it can generate *unique* solutions that cannot be computed by existing techniques.

Manually reported solutions. AuRUS can generate *equivalent* solutions to some manual repairs, reported in the literature, for the Arbiter, MinePump, RG2, and Lift cases. Additionally, it produces some *weaker/stronger* solutions, compared to the manual ones, as well as many unique solutions, giving further choices to the engineer in how to refine the specifications to get a realisable one.

Automatically generated solutions. We consider the work presented by Maoz et al. [41], limited to GR(1). They present two symbolic techniques for learning missing assumptions: *JVTS-Repair*, which generates new assumptions from the counter-strategies built as proofs of unrealisability; and *GLASS*, that computes safety, justice, and initial assumptions to ensure the corresponding safety, justice, and initial guarantees of the unrealisable specification. Also, [41] re-implements the algorithm presented in [8](*AMT13*), which also generates missing assumptions from counter-strategies. Notice that, while *GLASS* generates only 1 candidate repair, *JVTS-Repair* and *AMT13* may generate many candidates (because they try to remove the counter-strategies generated). To perform this comparison we took specifications from SYNTECH15 and others expressed in GR(1) such as RG1, RG2, and Lift (see Table 1).

Table 3: Repairs Overlapping

Case	AuRUS	GLASS	JVTS-Repair	AMT13
RG1	379 / 1	1 / 0	46 / 0	98 / 1
RG2	453 / 6	0 / 1	11 / 1	20 / 4
Lift	302 / 1	1 / 0	145 / 0	1 / 1
GyroV1	529 / 1	1 / 0	7 / 0	22 / 1
GyroV2	617 / 1	1 / 0	9 / 0	13 / 1
Humanoid458	582 / 0	1 / 0	1 / 0	1 / 0
Humanoid503	3 / 0	1 / 0	Timeout	Timeout
Humanoid531	86 / 0	1 / 0	Timeout	Timeout
Humanoid741	80 / 0	1 / 0	Timeout	Timeout
Humanoid742	99 / 0	1 / 0	3 / 0	Timeout
PCarV2-888	3 / 0	1 / 0	289 / 0	Timeout

Table 3 summarises, for each case, the number of unique solutions (left) produced by AuRUS, i.e., the number of solutions not generated by other techniques, and the number of equivalent solutions to one produced by other approaches (right). Notice that the comparison is always in between AuRUS and the related techniques, but we do not compare the related approaches against each other (i.e., we do not compare *GLASS* vs *JVTS-Repair* vs *AMT13*).

We observe that in most of the cases, AuRUS and *GLASS* complement each other, with the only exception for case RG2, where AuRUS produces an equivalent solution to the one proposed by *GLASS*. Moreover, for all considered cases, AuRUS generates many weaker/stronger repairs (between 2 and 80) than the one provided by GLASS, and the remaining are unique. When analysing *AMT13* repairs, we observe that AuRUS generates, for all cases in which the bound of *10 minutes* was not reached (same timeout of [41]) except for Humanoid458, between 1 to 4 equivalent solutions, and the remaining In the case of *JVTS-Repair*, AuRUS only generates 1 equivalent solution for the case RG2. In all case studies, many solutions generated by AuRUS maintains some relation (i.e., weaker/stronger) to the ones generated by JVTS-Repair and AMT13, but the majority are unique. The results evidence that the overlapping between the solutions is low, indicating that AuRUS can *complement* existing techniques, and provide a rich set of variants to the engineer to resolve the source of unrealisability.

6.4 Importance of the three properties

We first study AuRUS’ effectiveness, when some properties are deactivated from the fitness function. We run it under six extra configurations (see Fig. 3). In configurations (Syn, Sem, Syn+Sem) we deactivate the status checking, being AuRUS only guided by the syntactic and/or semantic similarity (realisability is only checked at the end of the execution, to check which candidate is a solution). In configurations (Real, Real+Syn, Real+Sem) we deactivate the syntactic and/or semantic similarity computation from the fitness function. Configuration Real+Syn+Sem denotes that AuRUS is guided by the three properties. Fig. 3 reports, for each configuration, the average percentage of repairs found per case study, with respect to the best performance previously discussed (with the three factors activated). Precisely, for the orange plots (repairs produced), the y-axis represents the relative difference between configuration runs and the best result for this metric across all configuration runs. For instance, the configuration *Real+Syn+Sem* produced on average 467 repairs while the highest number across all configurations is 531 in the arbiter example. In the red and green plots, the similarity is measured relative to the original specification. Notice that, by removing the realisability checking, AuRUS behaves pretty similar, or even worse, than random, affecting drastically its effectiveness in finding repairs. On the other hand, configurations that use realisability checking considerably improve AuRUS’ effectiveness, being able to find more solutions. In fact, we compute $\hat{A}_{12}$ values to compare different configurations for each case, and we can ensure that if AuRUS is guided by the 3 properties (*Real+Syn+Sem*), it obtains more repairs than the configurations that remove some of the properties.

We also analyse the quality of the best 10 ranked repairs, measured in terms of the syntactic (red) and semantic (green) similarity, that can be presented to the engineer for analysis and validation. It is almost always the case that our best-performing configuration (Real+Syn+Sem) obtains better syntactic and semantic similarities than the other configurations. Only in a few cases where random found more repairs than AuRUS, outperforms us in syntactical and semantic similarities. Two exceptions occur in Humanoid741/742, where the solutions found by configuration Real+Sem have better semantic similarities than the repairs found by Real+Syn+Sem. The outliers in Fig. 3 correspond to the mentioned cases.

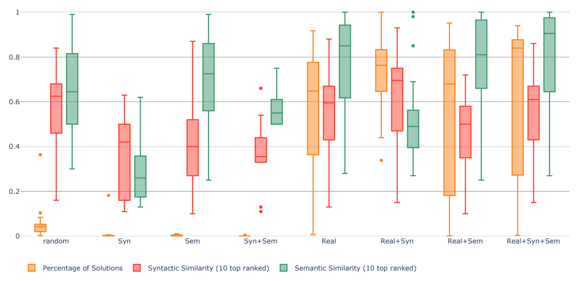


Figure 3: Impact of each factor of the fitness.

6.5 Evaluating our Model Counting Approach

To evaluate the precision and scalability of our model counting approach, we compare it with two related approaches. Firstly, we re-implemented an established exact model counting approach, that will be used as a baseline in the comparison. We basically encode LTL formulas as propositional formulas, such that, for a given bound k, each satisfying valuation of the encoding corresponds to a lasso-trace of k states of the formula [33]. Then, we can use a propositional model counter [28, 49, 50] to indirectly solve the LTL (bounded) model counting. The propositional encoding explodes exponentially as the bound is incremented, thus it can only be applied to small values of k. For the comparison, we additionally consider the approach that served us as motivation, presented in [19]. It also attempts to approximate LTL model counting, by generating a regex from a formula, then fed to the string model counter ABC [10] to estimate the number of models. Encoding our reactive specifications into propositional constraints can quickly produce formulas beyond what exact model counters can analyse. Therefore, we generate ten sets, $S_0, \ldots, S_9$, with 50 random LTL formulas, feasible for all approaches, taken from a well-established benchmark [1], used for assessing LTL SAT solvers [36]. For each formula in each set, we compute the number of models, for $k \in [6..10]$, using the *Exact* Model-Counting, the approach of [19] (*RE*), and our Approximate Model-Counting (*ApMC*). Then, we rank the formulas in ascending order w.r.t. the number of models obtained with each technique. We compare these rankings to verify if the techniques preserve the ranking of the exact MC.

Table 4: Model Counting results.

Set	Time Exact	Time ApMC	Diff Exact - ApMC	Time RE	Diff Exact - RE
$S0$	3685	2	0	102	11
$S1$	8815	2	2	102	9
$S2$	5757	2	0	101	7
$S3$	5299	2	0	101	6
$S4$	5844	2	0	102	5
$S5$	5665	2	0	102	6
$S6$	5130	2	0	103	10
$S7$	5929	2	0	103	8
$S8$	8800	2	0	101	10
$S9$	6172	2	0	103	7

Table 4 shows the results of the comparison only for bound $k = 10$ (other bounds are in the tool's site). It reports the execution time in seconds and the difference between rankings provided by *ApMC* and *RE* w.r.t. the ranking of the *Exact* model counter. The results show that *ApMC*, in 9/10 sets, produces the same ranking as *Exact*. In only 1 set it misclassified 2/50 formulas. Contrary, *RE* misclassified 8 formulas per set on average. *Exact* required 470 seconds, for a bound of 6, to 6000 seconds, for a bound of 10, to analyze each set on average. *RE* required more than 100 seconds per set, while *ApMC* required only 2 seconds per set. We assess scalability by testing approaches on 26 Table 1 specifications. *Exact* quickly becomes infeasible, while *RE* failed on 22/26 specifications, but *ApMC* succeeded on all, even with large bounds.

7 RELATED WORK

In reactive modeling, the system's expected properties are captured in temporal logics [38, 39], this is a typical setting in many essential activities, such as model checking [17], property monitoring [11], and model-based testing [26]. Detecting and finding flaws in specifications have been the focus on many studies [20, 21, 52] Recent works present different techniques to automatically repair the system's specification [6, 15, 18]. Unlike these approaches, which target the behavioural model, AuRUS aims to repair the declarative specification from which the synthesis tool will later generate an adequate model. LTL-Reactive synthesis has been studied for many years [7, 9, 24, 40, 45, 46]. Many approaches have focused on diagnosing the cause of unsynthesisability by computing a core of assertions that makes the specification unreal [48] or by generating a counter-strategy showing how the environment can prevent the controller from satisfying the guarantees [47]. Other approaches focus on undesirable properties of realisable specifications that affect the controllers' quality [22]. AuRUS guarantees to produce satisfiable and realisable repairs, and the mentioned techniques can complement the analysis to assess and improve our repairs' quality.

Recent approaches focus on inferring missing assumptions from unrealisable specifications [8, 13, 14, 37, 41]. Their limitations are twofold: they work on LTL fragments, e.g., GR(1), and only attempt to solve unrealisability, adding assumptions, not considering that existing ones/guarantees might be incorrect, which is often the case [5, 51, 52]. We show in Section 6.3 that AuRUS complements these techniques by being able to analyse general LTL specifications, changing both assumptions and guarantees, and providing more variants to repair unrealisability. Program repair tools, like GenProg [34] and DirectFix [42], use evolutionary algorithms to explore syntactical variants of the buggy program. These algorithms also aim to guide the search toward repairs similar to the input program. AuRUS uses LTL model counting to measure the semantic distance of the candidates concerning the initial specification. We show that existing LTL model counting tools [19, 25] quickly reach their scalability limits. Thus, we developed an approach to approximate it, proving that it is more efficient than the mentioned methods.

8 CONCLUSION

This paper presents AuRUS, a *search-based* approach to repair unrealisable specifications. Compared to previous methods that typically focus their analyses on identifying missing assumptions, AuRUS aims to modify assumptions and guarantees. The key aim is to generate realisable versions close to the given unrealisable ones. We defined syntactic and semantic similarity notions, essentials in guiding the algorithm towards quality solutions. AuRUS succeeded in repairing several case studies from the literature, established benchmarks, and managed to generate solutions in line with manual and automated fixes.

ACKNOWLEDGMENT

This work is made possible by the support of the Luxembourg National Research Funds (FNR) through the CORE project grant C19/IS/13646587/RASoRS, the PRODIGY Project (TED2021-132464B-I00) funded by MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR, as well as a research grant from Nomadic Labs and the Tezos Foundation.

REFERENCES

[1] Aalta benchmark. https://www.lab301.cn/aalta/node3.html.
[2] Commons math: The apache commons mathematics library. https://commons.apache.org/proper/commons-math/.
[3] The reactive synthesis competition. www.syntcomp.org.
[4] Synthesis competition repository. https://bitbucket.org/swenjacobs/syntcomp/.
[5] Dalal Alrajeh, Antoine Cailliau, and Axel van Lamsweerde. Adapting requirements models to varying environments. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 50–61, New York, NY, USA, 2020. Association for Computing Machinery.
[6] Dalal Alrajeh and Robert Craven. Automated error-detection and repair for compositional software specifications. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods*, pages 111–127, Cham, 2014. Springer International Publishing.
[7] R. Alur and S. La Torre. Deterministic generators and games for LTL fragments. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 291–300, June 2001.
[8] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Counter-strategy guided refinement of GR(1) temporal logic specifications. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 26–33, 2013.
[9] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. *IFAC Proceedings Volumes*, 31(18):447 – 452, 1998. 5th IFAC Conference on System Structure and Control 1998 (SSC'98), Nantes, France.
[10] 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.
[11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.
[12] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.
[13] Davide G Cavezza and Dalal Alrajeh. Interpolation-based GR(1) assumptions refinement. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 281–297. Springer, 2017.
[14] Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Environment assumptions for synthesis. In Franck van Breugel and Marsha Chechik, editors, *CONCUR 2008 - Concurrency Theory*, pages 147–161, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
[15] George Chatzieleftheriou, Borzoo Bonakdarpour, Panagiotis Katsaros, and Scott A. Smolka. Abstract model repair. *Log. Methods Comput. Sci.*, 11(3), 2015.
[16] 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.
[17] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.
[18] Renzo Degiovanni, Dalal Alrajeh, Nazareno Aguirre, and Sebastián Uchitel. Automated goal operationalisation based on interpolation and sat solving. In *ICSE*, pages 129–139, 2014.
[19] 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*, pages 1125–1135, 2018.
[20] Renzo Degiovanni, Facundo Molina, Germán Regis, and Nazareno Aguirre. A genetic algorithm for goal-conflict identification. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 520–531, 2018.
[21] 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.
[22] Nicolás D'Ippolito, Víctor A. Braberman, Nir Piterman, and Sebastián Uchitel. Synthesizing nonanomalous event-based controllers for liveness goals. *ACM Trans. Softw. Eng. Methodol.*, 22(1):9, 2013.
[23] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.
[24] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982.
[25] 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.
[26] Gordon Fraser, Franz Wotawa, and Paul Ammann. Testing with model checkers: a survey. *Softw. Test., Verif. Reliab.*, 19(3):215–261, 2009.
[27] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.
[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] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):563–583, 2013.
[30] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.
[31] Hadas Kress-Gazit and Hazem Torfah. The challenges in specifying and explaining synthesized implementations of reactive systems. In Bernd Finkbeiner and Samantha Kleinberg, editors, Proceedings 3rd Workshop on *formal reasoning about Causation, Responsibility, and Explanations in Science and Technology*, Thessaloniki, Greece, 21st April 2018, volume 286 of *Electronic Proceedings in Theoretical Computer Science*, pages 50–64. Open Publishing Association, 2019.
[32] Jan Kretínský, Tobias Meggendorfer, and Salomon Sickert. Owl: A library for ω-words, automata, and LTL. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, pages 543–550, 2018.
[33] 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.
[34] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38(1):54–72, January 2012.
[35] Jianwen Li, Geguang Pu, Lijun Zhang, Yinbo Yao, Moshe Y. Vardi, and Jifeng He. Polsat: A portfolio LTL satisfiability solver. *CoRR*, abs/1311.1602, 2013.
[36] Jianwen Li, Shufang Zhu, Geguang Pu, and Moshe Y Vardi. SAT-based explicit LTL reasoning. In *Haifa Verification Conference*, pages 209–224. Springer, 2015.
[37] Wenchao Li, Lili Dworkin, and Sanjit A Seshia. Mining assumptions for synthesis. In *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011)*, pages 43–50. IEEE, 2011.
[38] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.
[39] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety.* Springer-Verlag New York, Inc., New York, NY, USA, 1995.
[40] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, 1984.
[41] Shahar Maoz, Jan Oliver Ringert, and Rafi Shalom. Symbolic repairs for GR(1) specifications. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada*, pages 1016–1026, 2019.
[42] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 448–458, 2015.
[43] Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. Strix: Explicit reactive synthesis strikes back! In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, pages 578–586, 2018.
[44] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer, 1996.
[45] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380, 2006.
[46] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.
[47] Vasumathi Raman and Hadas Kress-Gazit. Explaining impossible high-level robot behaviors. *Trans. Rob.*, 29(1):94–104, February 2013.
[48] Viktor Schuppan. Towards a notion of unsatisfiable cores for LTL. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, pages 129–145, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
[49] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. Ganak: A scalable probabilistic exact model counter. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 1169–1176. International Joint Conferences on Artificial Intelligence Organization, 7 2019.
[50] 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.
[51] Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications.* Wiley, 2009.
[52] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Trans. Softw. Eng.*, 26(10):978–1005, October 2000.
[53] András Vargha and Harold D. Delaney. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.