SpecBCFuzz: Fuzzing LTL Solvers with Boundary Conditions

Luiz Carvalho
luiz.carvalho@uni.lu
SnT, University of Luxembourg
Luxembourg

Renzo Degiovanni
renzo.degiovanni@uni.lu
SnT, University of Luxembourg
Luxembourg

Maxime Cordy
maxime.cordy@uni.lu
SnT, University of Luxembourg
Luxembourg

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

Yves Le Traon
yves.letraon@uni.lu
SnT, University of Luxembourg
Luxembourg

Mike Papadakis
michail.papadakis@uni.lu
SnT, University of Luxembourg
Luxembourg

ABSTRACT

LTL solvers check the satisfiability of Linear-time Temporal Logic (LTL) formulas and are widely used for verifying and testing critical software systems. Thus, potential bugs in the solvers' implementations can have a significant impact. We present SpecBCFuzz, a fuzzing method for finding bugs in LTL solvers, that is guided by *boundary conditions* (BCs), corner cases whose (un)satisfiability depends on rare traces. SpecBCFuzz implements a search-based algorithm that fuzzes LTL formulas giving relevance to BCs. It integrates syntactic and semantic similarity metrics to explore the vicinity of the seeded formulas with BCs. We evaluate SpecBCFuzz on 21 different configurations (including the latest and past releases) of four mature and state-of-the-art LTL solvers (NuSMV, Black, Aalta, and PLTL) that implement a diverse set of satisfiability algorithms. SpecBCFuzz produces 368,716 bug-triggering formulas, detecting bugs in 18 out of the 21 solvers' configurations we study. Overall, SpecBCFuzz reveals: *soundness issues* (wrong answers given by a solver) in Aalta and PLTL; *crashes*, e.g., segmentation faults, in NuSMV, Black and Aalta; *flaky behaviors* (different responses across re-runs of the solver on the same formula) in NuSMV and Aalta; *performance* bugs (large time performance degradation between successive versions of the solver on the same formula) in Black, Aalta and PLTL; and no bug in NuSMV BDD (all versions), suggesting that the latter is currently the most robust solver.

KEYWORDS

Fuzzing, Search-Based Software Engineering, Linear-time Temporal Logic

ACM Reference Format:
Luiz Carvalho, Renzo Degiovanni, Maxime Cordy, Nazareno Aguirre, Yves Le Traon, and Mike Papadakis. 2024. SpecBCFuzz: Fuzzing LTL Solvers with Boundary Conditions. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639087

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639087

1 INTRODUCTION

LTL solvers are tools that automatically check the satisfiability of Linear-time Temporal Logic (LTL) formulas [56]. LTL solvers are typically used in software verification [6, 7, 38, 76, 77], software requirements analysis [28, 29], controller synthesis [10, 18], test generation and fuzzing [5, 43, 59], thereby playing a key role in the quality assurance of safety-critical systems.

Given that LTL solvers are used in the development of critical systems, their correctness and reliability are of paramount importance. For instance, when solvers are used as part of verification tools, incorrect solver results can leave large portions of programs under analysis unverified, with the unfortunate effect of increasing the risk of missing potential bugs and vulnerabilities. Thus, solvers are constantly engineered to fix bugs.

Testing LTL solvers is particularly challenging. The reason is that LTL solvers base their satisfiability analyses on the intrinsically complex temporal semantics of LTL formulas, which involves reasoning about infinite traces. Moreover, this complexity is increased since in order to efficiently produce solutions, solvers are continuously optimized via analytical and heuristic-based algorithms, e.g., via machine learning-based LTL SAT prediction [39, 53, 55] and other techniques. This gives rise to various types of bugs, such as erroneous results (wrong satisfiability responses), system crashes (solver aborts exceptionally, e.g., due to segmentation faults), flaky behaviors (different conclusions across different re-runs of the solver on the same formula), and performance issues (large performance divergences between successive versions of the solver on the same formula).

Moreover, checking the satisfiability of non-trivial LTL formulas often depends on corner cases (rare and subtle traces), that if solvers miss to explore, may lead to incorrect results. This makes LTL solver development challenging, and calls for the development of effective testing techniques, aiming to uncover the different types of bugs that can occur in LTL solving software. While fuzzing approaches exist for testing SAT and SMT solvers [31, 42], the problem of testing *LTL* solvers remains largely unexplored. To the best of our knowledge, there is no approach aiming at automatically testing LTL solvers beyond some benchmark sets of formulas [68]. Thus, there is a lack of principled methods to purposely test LTL solvers.

We fill this gap by proposing SpecBCFuzz, a fuzzing method for LTL solvers that is guided by the particularities of LTL semantics. The key idea is to generate formulas that are likely to drive

the solvers towards corner cases – a general principle that has been applied successfully in other testing areas, e.g., Boundary-Value Analysis [2]. In our context, corner cases are formulas whose (un)satisfiability depends on few and unique traces (cases that require a "global" analysis that force the solver to make a complete computation and cross-check of the entire formula). In contrast, formulas for which solvers can conclude their (un)satisfiability from multiple common traces are less interesting because they do not require an in-depth exploration of the entire formula semantics (in a sense offering multiple opportunities based on which one can determine unsatisfiability).

To generate corner cases, we rely on *goal divergences*, a concept originated in goal-oriented requirements engineering [74]. When a specification $\mathcal{S}$ is composed of a (satisfiable) conjunction of goals $\bigwedge_i G_i$ (each G_i is an LTL formula), a *divergence*, also known as a *boundary condition*, is a condition δ whose occurrence makes the specification inconsistent, i.e., its conjunction with the whole set of goals is unsatisfiable, while its conjunction with any strict subset of the goals remains satisfiable. As an example, consider a mine pump controller [45] with the following two goals: *"the pump shall be on when the water level is above the high threshold"*, and *"the pump shall be off when methane is detected in the mine"*. These can be formalized as $G_1 : \Box(hw \rightarrow p)$ and $G_2 : \Box(m \rightarrow \neg p)$, where *hm*, *m* and *p* stand for "high water", "methane" and "pump on", respectively. Although these goals are globally consistent, e.g., they are satisfiable in cases where methane is not detected in the mine, they are conditionally inconsistent (unsatisfiable) when the water level is high and methane is present at the same time. Thus, a boundary condition for G_1 and G_2 is $\delta : \Diamond(hm \wedge m)$.

Our approach works in two steps. First it considers a satisfiable (seed) formula $\mathcal{S}$ and produces a set of unsatisfiable formulas $\{\mathcal{S} \wedge \delta_i\}$ based on the set of boundary conditions δ_i for $\mathcal{S}$, automatically generated from $\mathcal{S}$. Then it generates mutated versions $\mathcal{S}'$ of the formula $\mathcal{S}$ based on which a set of potentially unsatisfiable formulas $\{\mathcal{S}' \wedge \delta_i\}$ are produced. Thus, our aim, given an original seed formula, is to explore its vicinity, i.e., formulas that are close to the original, together with the vicinity of the possible divergences of the formula, as illustrated in Figure 1. Starting from a satisfiable formula $\mathcal{S}$, SpecBCFuzz produces a set of unsatisfiable formulas $\{\mathcal{S} \wedge \delta_i\}$ by computing a set of boundary conditions δ_i for $\mathcal{S}$ and then explores their nearby solutions (shaded areas).

We claim that *boundary conditions are good for triggering faults in LTL solvers* because they include two important (semantic) properties: i) the conjunction of $\mathcal{S}$ with δ moves the specification from the satisfiable plane to the unsatisfiable plane thereby further challenging the solvers; ii) it forces the solvers to consider all goals *and* the boundary condition to prove the unsatisfiability of the formula (divergence definition property). Overall, boundary conditions force the solvers to perform an in-depth exploration that has a good potential to trigger bugs as shown by our results.

Proper fuzzing also requires the selection of diverse seeds. This brings two challenges for SpecBCFuzz: 1) the selection of formulas that have many divergences and 2) the computation of boundary conditions. In our analysis we used 25 specifications from the literature and computed a total of 346 boundary conditions, an average of 13.84 boundary conditions per specification. To efficiently produce tests, SpecBCFuzz relies on a search-based algorithm that explores

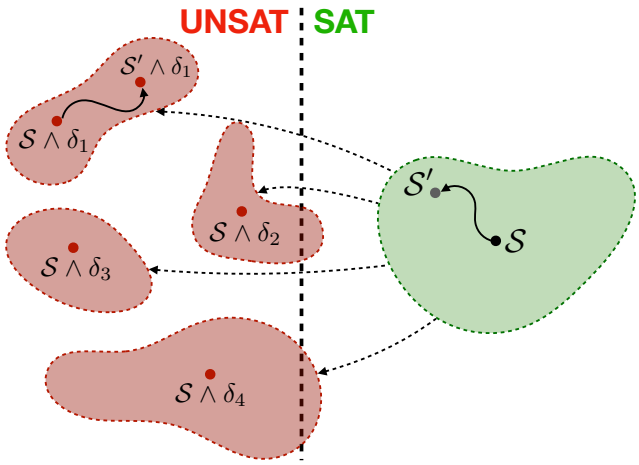

Figure 1: Exploring the vicinity of the divergences.

the vicinity of $\mathcal{S}$ and finds other formulas S' for which the boundary conditions $\Delta = \{\delta_i\}$ (previously computed from $\mathcal{S}$) remain relevant. To increase the likelihood that δ_i is also a boundary condition for S', the search is guided to preserve S' as similar as possible to S.

SpecBCFuzz implements a Non-dominated Sorted Genetic Algorithm (NSGA-III) [26] to evolve and mutate the original specification $\mathcal{S}$, guided by a multi-objective fitness function. Within this function, two similarity metrics – one *syntactic* and one *semantic* – compare the seeded formula $\mathcal{S}$ and the mutated formula $\mathcal{S}'$. SpecBCFuzz uses the Levenshtein edit distance [52] to measure syntactic similarity and an LTL model counting heuristic [13] as a semantic-related metric. To effectively explore the search space close to $\mathcal{S}$, we also consider an additional objective: to explore as many combinations as possible of the divergences captured by boundary conditions in Δ. This objective aims at exploring a wide spectrum characterized by the divergences from the sat plane to the unsat one.

Once the mutated formulas have been generated, SpecBCFuzz conjuncts them with the previously computed boundary conditions (yielding $\mathcal{S}' \wedge \delta_i$) as inputs for fuzzing the LTL solvers. To bypass the oracle problem, SpecBCFuzz cross-checks the answers given by different solvers (when looking for soundness bugs), different versions of the same solver (for performance bugs), and different runs of the same version on the same formula (for flakiness bugs).

To evaluate SpecBCFuzz, we conducted experiments on multiple releases (including the latest) of 4 LTL solvers under different settings. Our evaluation comprises: two variants of the *Aalta* (v1 and v2) solver based on tableaux [50, 51]; three variants of the *PTLT* (multipass-based and graph-based) solver, two based on tableaux [69, 78] and one based on BDDs [34, 58]; *BLACK*, based on tableaux [36, 37]; and two variants of *NuSMV* (v2.4.3, v2.5.4, v2.6.0), one based on BMC [20] and one based on BDDs [19]. In total, our evaluation involves 21 solvers' configurations.

Our experimental analysis uses as seeds 25 requirements specifications collected from the literature [28], for which a set of boundary conditions are automatically computed. *In total, SpecBCFuzz generates 368,716 bug-triggering formulas, revealing bugs in 18 out*

of the 21 solver configurations (except in the 3 configurations of NuSMV BDD). Furthermore, SpecBCFuzz reveals:

- Soundness issues (wrong solving results) in Aalta (v1 & v2) and PLTL (BDD).
- Crashes in 10 configurations of three solvers (NuSMV BMC, Black & Aalta).
- Flaky behavior in NuSMV BMC (3 versions of this solver) and Aalta (v1 & v2).
- Performance bugs in 7 versions of three solvers (Black, Aalta, and PLTL).
- No bugs in NuSMV BDD (all versions), suggesting that this is currently the most robust solver.

Our results also demonstrate that *all search objectives of SpecBCFuzz (the use of boundary conditions, the syntactic and semantic similarity metrics) contribute in finding bug triggering LTL formulas.* Finally, our analysis shows that *SpecBCFuzz outperforms a typical grammar-based fuzzer*, that was specifically implemented and tuned for fuzzing LTL solvers.

2 BACKGROUND

2.1 LTL, SAT and Model Counting

Linear-time Temporal Logic (LTL) is a formalism used for formal property specification of reactive systems [56]. Several formal methodologies, e.g., KAOS [73], have adopted LTL to express requirements [73] and perform analyses.

LTL assumes a linear topology of time, i.e., each instant is followed by a unique future instant, and LTL formulas are evaluated over infinite traces, that can be interpreted as system executions. Let AP be a set of propositional variables. LTL formulas are inductively defined using the standard logical connectives and temporal operators $\bigcirc$ and $\mathcal{U}$, as follows: *(i)* constants *true* and *false* are LTL formulas; *(ii)* every $p \in AP$ is an LTL formula; and *(iii)* if φ_1 and φ_2 are LTL formulas, then so are $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2$, $\bigcirc\varphi_1$ and $\varphi_1 \mathcal{U} \varphi_2$.

LTL formulas are evaluated over infinite traces of the form $\sigma = s_0\ s_1 \ldots$, where each s_i is a propositional valuation on 2^{AP}, i.e., $\sigma \in 2^{AP^\omega}$. Formulas with no temporal operators are evaluated in the first valuation of a trace. Given a trace σ, $\bigcirc\varphi$ is true in σ if and only if φ is true in $\sigma[1..]$ (the trace obtained by removing the first valuation from σ), and $\varphi_1 \mathcal{U} \varphi_2$ is true in σ if and only if there is a position i such that φ_2 holds in $\sigma[i..]$, and for all $0 \leq j < i$, φ_1 holds in $\sigma[j..]$. We consider the typical definitions for the operators $\Box$ (always), $\Diamond$ (eventually) and $\mathcal{W}$ in terms of $\bigcirc$, $\mathcal{U}$ and logical connectives.

An LTL formula φ is *satisfiable* (SAT) iff there exists at least one trace satisfying φ; otherwise, it is *unsatisfiable* (UNSAT).

Model counting computes the number of traces that satisfy a formula. Since LTL formulas are defined over infinite traces, this involves the computation of the number of canonical *finite* representations of *infinite* traces, such as lasso traces, as also done in bounded model checking [8]. Since computing the exact number of lasso traces is expensive [32], Brizzio et. al [13] proposed a bounded model counting approximation to compute the number of lasso traces satisfying an LTL formula, based on symbolic LTL automata representation and matrix multiplication. Thus, given an LTL formula φ and a bound k, we denote by $\#\textsc{Approx}(\varphi, k)$, the approximated number of lasso traces of length k satisfying φ. SpecBCFuzz employs #Approx model counting to compare the semantics of LTL formulas.

2.2 LTL Solvers

LTL satisfiability is a *decidable* problem [67], and tools implementing such checking are called LTL (SAT) solvers. Existing LTL solvers are designed to be efficient [68], to support diverse temporal operators [37], and to be expressive [48]. Some of the best known techniques for LTL solving include those based on bounded model checking (BMC), on binary decision diagrams (BDDs), on Tableaux, and on automata-theoretic approaches.

Bounded Model Checking (BMC) encodes LTL formulas as propositional formulas [8, 24, 49], for a given bound k. *Each* satisfying valuation of the encoding corresponds to a lasso trace of k states. NuSMV [19] implements a traditional BMC algorithm.

BDD-based Model Checking employs a symbolic representation of LTL formulas as binary decision diagrams (BDD), based on the formula's elementary sub-formulas [21, 23]. The BDD is built by applying rules that capture the semantics of the LTL operators. The satisfying instances are obtained by traversing paths in the BDD. NuSMV [19] and PLTL implement a traditional BDD-based SAT algorithm.

Tableau is a well-known logical satisfiability approach, based on the decomposition of the formula being assessed according to the semantics of its logical operators, to search for satisfying valuations. While a tableau for a positional formula is a finite tree capturing its semantics, a tableau for an LTL formula is a *graph* capturing the semantics of temporal operators in the formula.

There exist different particular variants of the process to build the tableau structure, classified as the so-called One-pass and Multi-pass tableaux methods. Among these, PLTL [69] implements the Schwendimann method, while BLACK [36, 37] implements the Reynolds method [66].

Automata-based approaches encode LTL formulas into automata, and then check for the emptiness of the languages corresponding to such automata [47, 75]. Different automata representations are used to improve the performance of solvers, such as the alternating automata (more efficient and succinct than other kinds of automata) [47, 75]. Additional heuristics are also used. For instance, Aalta (v.1) implements an on-the-fly automaton exploration [50].

2.3 Fuzzing

Fuzzing is a technique that generates input strings [60], used for testing activities [80] such as system and integration testing, semantic-oriented testing [64, 71], and security testing [9, 33].

Coverage-guided fuzzers, such as the American Fuzzy Lop (AFL) and its variants, implement various fuzzing strategies. For instance, AFLFast implements grey-box fuzzing guided by code coverage [16]; AFLGo implements a simulation annealing search guided by an inter-procedural distance measure [12]; and Zest implements a property-based fuzzing mechanism [63, 64].

Grammar-based Fuzzing typically takes a (context-free) grammar that describes the shape of the inputs, and produces random

inputs by traversing the grammar production rules. These are commonly used for testing compilers. Their objective is to efficiently produce many and diverse inputs, increasing the chances of triggering syntactic or semantic bugs.

Seeds are bootstraps of the bug finding process [40]. Good seeds are typically collected from a large number of representative cases or domain-specific scenarios, and have some meaningful semantics for the software under test. For instance, seeds can be built by crawling the internet [40], or can be user provided [59, 61]. Collected seeds are exploited by mutation and search-based strategies for generating semantically meaningful inputs.

Mutation Fuzzing introduces small changes to the given seeds with the aim of triggering additional behaviors in the target system. Preferably, the mutations should maintain the syntactic validity of the input. Common mutation operators remove, insert, and flip single elements of the inputs.

Search-based Fuzzing leverages mutation fuzzing by also guiding the seeds selection and evolution with one or more fitness functions [65]. The optimized meta-heuristics are domain-specific for the target system. Typical fitness metrics focus on the code coverage and structure of the generated inputs.

In this work, we present SpecBCFuzz, a search-based fuzzing approach that takes LTL specifications with boundary conditions as seeds. SpecBCFuzz implements a set of evolutionary operators (mutation and crossover) to evolve LTL formulas, and two similarity metrics (one syntactic and one semantic) to search in the vicinity of the given seeds, with the aim of finding critical bugs, e.g., soundness issues, crashes, and flakiness problems, in LTL solvers.

3 THE SPECBCFUZZ APPROACH

Figure 2 shows an overview of SpecBCFuzz. It follows four steps to generate and evolve LTL formulas for testing LTL solvers. Firstly, given a seeded formula $\mathcal{S}$ (a goal-oriented requirements specification), SpecBCFuzz computes a set $\Delta = \{\delta_1, \ldots, \delta_k\}$ of boundary conditions that capture divergences in $\mathcal{S}$.

The second step focuses on evolving $\mathcal{S}$ into other LTL formulas $\mathcal{S}'$ that are likely to also be divergent with respect to the boundary conditions in Δ. To do so, SpecBCFuzz implements a multi-objective search algorithm (more precisely, NSGA-III [26]) that applies genetic operators (mutation and crossover) to produce new formulas. The multi-objective function SpecBCFuzz relies on is driven by formula similarity metrics; it seeks to increase the likelihood of the new formulas to be divergent with respect to the boundary conditions computed from the seeded specification $\mathcal{S}$.

Once a formula $\mathcal{S}'$ has been produced, it is conjoined with each δ_i to form a new set $\{\mathcal{S}' \wedge \delta_i\}_i$ that SpecBCFuzz tests each solver with (step 3). In our experiments we considered 21 configurations of four mature and state-of-the-art LTL solvers, including their corresponding latest versions, namely, NuSMV, Aalta, PLTL and Black. Of course, other solvers and future versions of the considered solvers may be easily integrated in the future.

To bypass the oracle problem, SpecBCFuzz relies on differential testing to cross-check the behavior of the different solvers and runs (step 4). More precisely, SpecBCFuzz looks for *soundness* bugs (different solvers giving different satisfiability answers), *crashes* (a solver aborts the execution exceptionally), *flakiness* (different runs of the same solver version on the same formula yields different answers), and performance bugs (large performance differences between different versions of the same solver on a same given formula).

In what follows, we detail the above-mentioned steps and, in particular, how the evolutionary search process and the solver testing steps intertwine.

3.1 Computing Divergences

In goal-oriented methodologies, e.g., KAOS [73], requirements are organized as a set of domain properties (*Dom*) and a set of goals ($\{G_i\}$). Intuitively, the goals are the properties we expect the system to achieve, while domain properties capture assumptions and descriptive statements of the environment, e.g., physical or normative laws. Since requirements descriptions can be ambiguous and incomplete, and different stakeholders may have different expectations from the system, specified goals can contradict one another (i.e., they can be conflicting) [73, 74]. If there is a strong conflict between the goals and the domain properties, they cannot be satisfied together ($Dom \wedge (\bigwedge_i G_i) \models false$), and the specification is said to be *inconsistent*.

For consistent goals, there exists a weaker form of conflict, named *divergence* [73, 74]. It represents a condition whose occurrence makes the goals inconsistent (i.e., they cannot be satisfied under the condition that the divergence occurs). Formally, a set $G = \{G_1, \ldots, G_n\}$ of goals is *divergent* with respect to *Dom* if there exists a *boundary condition* (a formula) δ such that the following conditions hold together:

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

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

$$\delta \not\equiv \neg(G_1 \wedge \ldots \wedge G_n) \qquad (\textit{non-triviality})$$

The first condition establishes that, when δ holds, the whole set of goals cannot be simultaneously satisfied. The second condition states that, if any of the goals are disregarded, then consistency is recovered. Also, it prevents δ from being *false*, since it has to be consistent with the domain *Dom*. The third condition prohibits a boundary condition to be simply the negation of the goals.

SpecBCFuzz relies on divergences and their corresponding boundary conditions to create LTL formulas that force solvers to make in-depth analyses to conclude about satisfiability, leaning on the three properties of boundary conditions. The rationale is that, since δ semantically connects all the goals of the formula, it can complicate the variable splitting heuristics implemented by the solvers, forcing them to perform a more exhaustive search.

To have meaningful formulas, SpecBCFuzz uses requirements specifications from the literature as seeds, and computes their boundary conditions using the automated approach by Degiovanni et al. [28]. However, because this approach and its alternatives [29, 54] are computationally expensive, their application throughout the search process (i.e., on all formulas $\mathcal{S}'$ that SpecBCFuzz generates in the vicinity of $\mathcal{S}$) would require prohibitively expensive computation time (in our experiments, the approach in [28] requires 2,183 seconds for computing boundary conditions, on average). This is a clear obstacle for fuzzing, that is based on fast

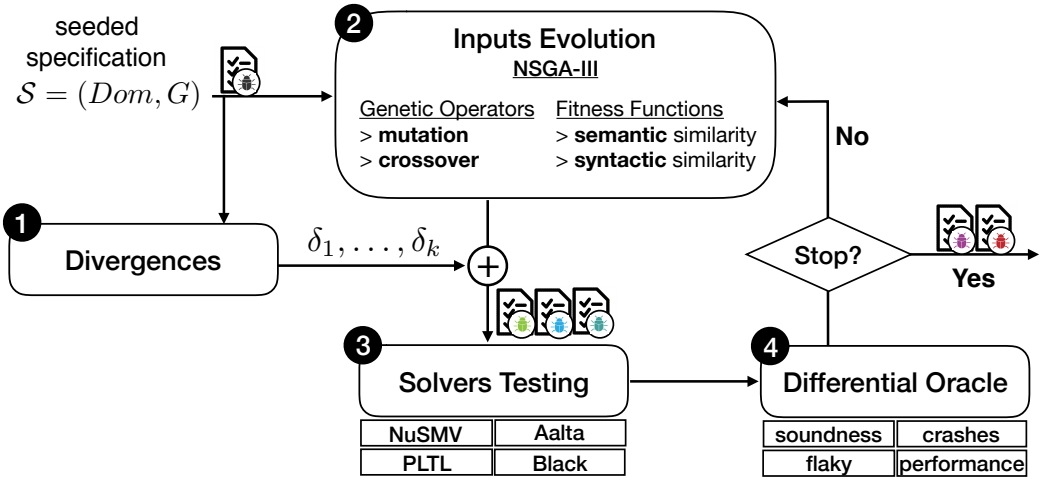

Figure 2: Evolutionary search implemented by SpecBCFuzz.

generation of tests. Therefore, we only generate boundary conditions on the original specifications/formulas (seeds).

3.2 Search Process

Algorithm 1 describes how SpecBCFuzz combines and intertwines the evolutionary search steps (NSGA-III) and the on-the-fly testing of the LTL solvers.

Given seeded formula $\mathcal{S} = (Dom, G)$, SpecBCFuzz starts by computing a set Δ of boundary conditions capturing $\mathcal{S}$'s divergences (Line 2). In Line 3, it initializes the sets where the different kinds of bug-triggering inputs will be saved, and in Line 4 it initializes the set of candidate inputs (population) I, i.e., the NSGA-III set of non-dominated individuals, with the seeded formula $\mathcal{S}$.

From Lines 5-16, the algorithm relies on specific features of NSGA-III, such as the prioritization, selection and evolution of individuals (formulas) for fuzzing the LTL solvers, until a termination condition is met, e.g., a specific execution time or number of generations is reached. Particularly, in Line 6, the algorithm picks C, one of the non-dominated candidate formulas produced so far. Then, in Line 7, SpecBCFuzz applies the implemented genetic operators to C, i.e., the mutation and cross-over especially designed for manipulating LTL formulas (see Section 3.3), and iterates on each generated formula $\mathcal{S}'$.

In this inner loop, for each boundary condition $\delta \in \Delta$, SpecBCFuzz generates a candidate bug-triggering formula i by combining it with the mutated formula, yielding $\mathcal{S}' \wedge \delta$ (Line 9). Then, it invokes each solver under test and gathers their corresponding outputs, i.e., $o_j = Solver_j(i), 1 \leq j \leq N$ (Line 10). SpecBCFuzz analyses the outputs and, whenever a bug is detected, it saves relevant information useful to reproduce it later, e.g., the solver's name, version and configuration as well as the bug-triggering formula i. In Line 13, SpecBCFuzz computes the multi-objective fitness value for the mutated formula $\mathcal{S}'$ (see Section 3.4). Finally, SpecBCFuzz updates the set I of non-dominated individuals according to the just computed fitness values for $\mathcal{S}'$, which can survive for the next generations.

Algorithm 1 takes a formula $\mathcal{S} = (Dom, G)$ and its divergences as input and returns correctness, crashes, flaky and performance $\langle \mathcal{B}_s, \mathcal{B}_c, \mathcal{B}_f, \mathcal{B}_p \rangle$ issue-triggering inputs.

```
1:  function SpecBCFuzz(S, BCs): ⟨B_s, B_c, B_f, B_p⟩
2:    Δ ← ComputeBCs(S)                          ▷ Set of boundary conditions
3:    B_s, B_c, B_f, B_p ← ∅, ∅, ∅, ∅             ▷ Set of bugs found
4:    I ← {S}                                    ▷ Set of evolved formulas
5:    while I ≠ ∅ and ¬(termination-criteria) do
6:      C ← PickBest(I)   ▷ NSGA-III non-dominated individual
7:      for all S' ∈ Evolve(C) do   ▷ NSGA-III genetic operators
8:        for all δ ∈ Δ do
9:          i ← S' ∧ δ                ▷ candidate bug-triggering input
10:         ⟨o_1, ... o_N⟩ ← Solver_1(i), ..., Solver_N(i)
11:         B_s, B_c, B_f, B_p ← UpdateBugs(i, o_1, ..., o_N)
12:       end for
13:       Fitness(S, S', Δ, o_1, ... o_N)▷ NSGA-III multi-objectives
14:       I ← UpdateBest(I, S')          ▷ NSGA-III prioritization
15:     end for
16:   end while
17:   return ⟨B_s, B_c, B_f, B_p⟩
18: end function
```

In the remainder of this section, we present the details regarding the genetic operators and the multi-objective fitness computation.

3.3 LTL Genetic Operators

SpecBCFuzz implements the standard mutation and cross-over operators for LTL formulas [28, 54].

To illustrate these operators, consider the formula $\mathcal{F} : \Box(p \rightarrow \neg q \wedge r)$. Figure 3 shows four examples of mutations that SpecBCFuzz applies. For instance, mutant M_1 replaces the unary operator $\Box$ in $\mathcal{F}$ by $\Diamond$; mutant M_2 replaces proposition p by r; in M_3, binary operator $\rightarrow$ is replaced by $\vee$; while mutant M_4 removes part of the formula. All mutations are guaranteed to respect the LTL syntax,

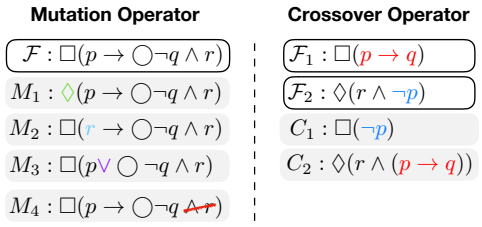

Figure 3: Mutation and Crossover operators.

and can produce any LTL formula in the language, strictly used in the seeded formula $\mathcal{S}$, i.e., no new atomic proposition is added.

Crossover operator takes two goals $\mathcal{F}_1$ and $\mathcal{F}_2$ from the same individual C (or from a different individual previously generated), and swaps two sub-formulas randomly selected from these goals, yielding two new formulas C_1 and C_2. For example, given $\mathcal{F}_1 : \square(p \rightarrow q)$ and $\mathcal{F}_2 : \Diamond(r \wedge \neg p)$ as in Figure 3, the crossover operator might select sub-formula $\alpha : p \rightarrow q$ from $\mathcal{F}_1$ and sub-formula $\beta : \neg p$ from $\mathcal{F}_2$. Then, it proceeds to swap the sub-formulas by replacing α by β in $\mathcal{F}_1$, and vice versa in $\mathcal{F}_2$, leading to two new formulas $C_1 : \square(\neg p)$ and $C_2 : \Diamond(r \wedge (p \rightarrow q))$. This operator guarantees by construction to produce syntactically valid LTL formulas.

3.4 Multi-objective Fitness

For each candidate variant $\mathcal{S}'$ generated by the application of the genetic operators, SpecBCFuzz computes the fitness value for the three objectives that guide the search (Line 13 in Algorithm 1). Since computing a new set of boundary conditions for each candidate is practically infeasible (requiring on average 2,183 seconds [28]), these three objectives aim at driving SpecBCFuzz's search process towards formulas $\mathcal{S}'$ for which the boundary conditions Δ capture divergences in $\mathcal{S}'$ as well. These three objectives are: semantic similarity between $\mathcal{S}'$ and $\mathcal{S}$, their syntactic similarity, and the approximated number of boundary conditions that remain relevant for $\mathcal{S}'$.

The rationale behind semantic similarity is that two semantically close formulas have a higher likelihood to have boundary conditions in common. Thus, our semantic similarity function, denoted by $SemSim(\mathcal{S}, \mathcal{S}')$, computes the ratio between the number of behaviors common to $\mathcal{S}'$ and $\mathcal{S}$ over the union of their behaviors. Since computing the set of lasso traces of an LTL formula is computationally prohibitively expensive, SpecBCFuzz instead relies on an efficient model counting heuristic [13], which approximates the number of accepted lasso traces of any formula (cf. Section 2.1). Hence, given a bound k for the lasso traces, the semantic similarity between $\mathcal{S}$ and $\mathcal{S}'$ is computed as:

$$SemSim(\mathcal{S}, \mathcal{S}') = \frac{\#\textsc{Approx}(\mathcal{S} \wedge \mathcal{S}', k)}{\#\textsc{Approx}(\mathcal{S} \vee \mathcal{S}', k)}$$

where $\#\textsc{Approx}(\mathcal{S} \wedge \mathcal{S}')$ is the approximated number of accepted lasso traces for $\mathcal{S} \wedge \mathcal{S}'$ (intersection) and $\#\textsc{Approx}(\mathcal{S} \vee \mathcal{S}')$ is the approximate number for $\mathcal{S} \vee \mathcal{S}'$ (union). Small values for $SemSim(\mathcal{S}, \mathcal{S}')$ indicate that the behaviors described by $\mathcal{S}'$ deviate too much from those described by the seeded formulas $\mathcal{S}$ as it has few behaviors in common. If this value gets closer to 1 it means that the two formulas share most of their corresponding behaviors. In the case where $\mathcal{S}'$ is unsat or contradicts $\mathcal{S}$, we set the semantic similarity to 0 to discard this unsatisfiable formula in subsequent iterations.

Syntactic similarity, denoted by $SynSim(\mathcal{S}, \mathcal{S}')$, is another objective we use to further support semantic similarity. To measure it, SpecBCFuzz uses Levenshtein distance [52] to compute the distance between the text representations of the formulas. The Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one word into the other. Hence, $SynSim(\mathcal{S}, \mathcal{S}')$ is computed as the ratio between the number of tokens changed from $\mathcal{S}$ to obtain $\mathcal{S}'$ among the maximum number of tokens corresponding to the largest formula ($maxLength = max(length(S), length(cR))$). Specifically:

$$SynSim(\mathcal{S}, \mathcal{S}') = \frac{maxLength - Levenshtein(\mathcal{S}, \mathcal{S}')}{maxLength}$$

Since SpecBCFuzz aims at generating formulas for which boundary conditions in Δ remain relevant, it uses a simple heuristic to count for the number of boundary conditions $\delta \in \Delta$ that remain unsatisfiable with respect to $\mathcal{S}'$:

$$\#BCs(\mathcal{S}', \Delta) = \frac{k - \sum_{j=1}^{k} isUNSAT(\mathcal{S}', \delta_j)}{k}$$

where $isUNSAT(\mathcal{S}', \delta_j)$ will check if the majority of the solvers' outputs $o_1, \ldots, o_N$ indicate that formula $\mathcal{S}' \wedge \delta_j$ is unsat. Taking the majority as the correct answer, will help SpecBCFuzz to be robust in the cases where formula $\mathcal{S}' \wedge \delta_j$ triggers a bug in a solver.

Overall, the three objectives will guide SpecBCFuzz to search in the vicinity of $\mathcal{S}$, with high chances to be divergent with respect to the boundary conditions in Δ, making them good candidates for triggering bugs in LTL solvers.

3.5 Differential Testing

SpecBCFuzz, inspired by differential testing, defines simple oracles to detect four kinds of bugs. Given the input formula $i = \mathcal{S}' \wedge \delta_j$ and solvers' outputs $o_1, \ldots, o_N$, SpecBCFuzz first computes the number of $\#sat$ ($\#\{o \in o_1, \ldots, o_N \mid o = SAT\}$) and $\#unsat$ ($\#\{o \in o_1, \ldots, o_N \mid o = UNSAT\}$) responses. Then, the expected outcome is defined as follows:

$$Expected(\mathcal{S}' \wedge \delta_j) = \begin{cases} sat & \#sat > \#unsat \\ unsat & \#unsat > \#sat \\ unknown & \text{otherwise} \end{cases}$$

For example, let us assume that $o_1 \neq Expected(\mathcal{S}' \wedge \delta_j)$, i.e., $Solver_1$ produces an output different from the expected one. SpecBCFuzz will first re-run $Solver_1(\mathcal{S}' \wedge \delta_j)$ 100 times, in order to confirm that the solver is consistently producing the same output for the given input. If the solver is always producing the same unexpected output, then the bug is confirmed and information regarding the solver and the bug-triggering formula are added to the corresponding set. Precisely, if o_1 is sat/unsat, i.e., the solver produces an incorrect output, this is a soundness bug and the bug-info is added to the set $\mathcal{B}_s$; otherwise, if o_1 is unknown because the solver always crashes with the same input, the bug-info is added to $\mathcal{B}_c$.

In the case that, after re-running multiple times the same solver with the same input, it produces a different output compared to

the firstly observed o_1, e.g., o_1 was sat, but when re-run it often produced unsat, then we consider this behavior as flaky and we add the bug info to the set $\mathcal{B}_f$.

Finally, for each input formula, we compute the average execution time T of the solvers for producing a valid (sat/unsat) outcome. Then, when performing the re-runs, if a solver takes more than 300 times the average execution time in producing the output (i.e., the execution time is greater than $300 \times T$), or it reaches a predefined timeout of 24 hours (just for the re-runs), then SpecBCFuzz will consider this as a potential performance bug and warn the tester by adding the corresponding bug-info to the set $\mathcal{B}_p$. After finalizing the fuzzing campaign, SpecBCFuzz returns the sets of the identified bug-triggering inputs.

4 EXPERIMENTAL SETUP

4.1 Research Questions

We investigate the following research questions:

- **RQ1:** How effective is SpecBCFuzz in revealing bugs in LTL solvers? How robust are the solvers?
- **RQ2:** How does each objective of the fitness function contribute to SpecBCFuzz' effectiveness?
- **RQ3:** Do boundary conditions and vicinity exploration provide effective guidance to reveal LTL solver bugs?

RQ1 evaluates the effectiveness of our approach in revealing bugs in the studied LTL solvers. Thus, we execute SpecBCFuzz against 21 configurations of four solvers (see Section 4.3). First, we report the number of soundness, crashes, flakiness, and performance issues that were triggered by the formulas produced by our fuzzing campaigns. We then perform an analysis on the bug-triggering inputs to categorize the inconsistent executions into more general buggy patterns, i.e., symptoms we observe on one or more specific versions of a solver. Intuitively, two formulas f_1 and f_2 are in the same cluster, if they trigger the same buggy symptom, e.g., a crash with the same error message, in the same version of the same solver. Thus, a buggy pattern corresponds to one or more concrete bugs to fix in the concerned solver versions.

To answer RQ2, we conduct an ablation study to quantify the contribution of each objective of SpecBCFuzz's fitness function to its effectiveness. Thus, we disable each objective (separately) and compare the number of bug-triggering formulas (for each previously identified buggy pattern) that SpecBCFuzz reports.

RQ3 aims to validate the two main hypotheses that SpecBCFuzz relies on, i.e., 1) specifications with boundary conditions are good seeds for revealing bugs, and 2) exploring the local vicinity of the original formula is effective in triggering faults. A superior effectiveness of SpecBCFuzz would validate our hypotheses and the principles our approach relies on. Hence, in addition to SpecBCFuzz (which uses the set $\{\mathcal{S} \wedge \delta_i\}_i \cup \{\mathcal{S}'_j \wedge \delta_i\}_{i,j}$ as seeds), we repeat our experiments on multiple baselines:

- SpecBCFuzz using $\{\mathcal{S} \wedge \delta_i\}_i$ as seeds (boundary conditions without vicinity exploration);
- SpecBCFuzz using $\{\mathcal{S}\} \cup \{\mathcal{S}'_j\}_j$ as seeds (vicinity exploration without boundary conditions);
- a probabilistic grammar-based fuzzing approach we fine-tuned to generate random LTL formulas, i.e., broad exploration without boundary conditions;
- a large benchmark[1] of 3,723 LTL formulas, of different complexities, used for studying the performance of LTL solvers [68].

4.2 Seeds

Our evaluation considers a total of 25 LTL formulas collected from the literature and different benchmarks. These formulas were previously used by several approaches for the identification and resolution of divergences [1, 17, 27–29, 54, 74]. Table 1 summarizes the number of LTL formulas of each seeded specification (#$\mathcal{S}$) and the number of boundary conditions (#Δ) computed with the approach by Degiovanni et al. [28].

Table 1: Seeded LTL formulas and divergences.

Specification	#$\mathcal{S}$	#Δ
minepump	3	14
simple arbiter-v1	4	28
simple arbiter-v2	4	20
prioritized arbiter	7	11
arbiter	3	20
detector	2	15
ltl2dba27	1	11
round robin	9	12
tcp	2	11
atm	3	24
telephone	5	4
elevator	2	3

Specification	#$\mathcal{S}$	#Δ
rrcs	4	14
achieve-avoid pattern	3	16
retraction pattern-1	2	2
retraction pattern-2	2	10
RG2	2	9
lily01	3	5
lily02	3	11
lily11	3	5
lily15	3	19
lily16	6	38
ltl2dba theta-2	1	3
ltl2dba R-2	1	5
simple arbiter icse2018	11	20

4.3 Considered Solvers

We aim at assessing the effectiveness of SpecBCFuzz in finding bugs in LTL solvers that implement a diverse set of algorithms and heuristics, where the state representation is symbolic or concrete, and the search is automata-based, tableaux, or propositional solving guided. Because of that, in addition to the latest version of each solver, we also consider some past versions that can potentially reveal different kinds of buggy behaviors.

In total we use 21 solvers' configurations in the evaluation, summarized in Table 2.

Particularly, **NuSMV** is one of the most adopted solvers for analyzing LTL requirements [19]. In total, we consider 6 configurations for NuSMV, including two SAT algorithms (BMC and BDD) for the latest (2.6.0) and past versions (2.5.4, 2.4.3).

Black [36, 37] is a recent solver that implements a one-pass tree-shaped tableau, recently proposed by Reynolds [66]. For producing satisfying instances, or proof of unsatisfiability, Black relies on propositional or SMT solvers such as MathSAT [22], Z3 [25], cvc5 [4], CryptoMiniSAT [70] and MiniSAT [30]. In total, we consider 10 configurations for Black, including the latest (0.9.2) and two recent (0.7.4, 0.5.2) versions, and their integration with MathSAT, Z3, cvc5 (supported only in v0.9.2) and CryptoMiniSAT. We had

[1] http://www.schuppan.de/viktor/atva11/

issues when Black uses MiniSAT as the back-end solver and thus, it was discarded.

Aalta is a concrete-state representation satisfiability algorithm that builds the automata capturing LTL formulas on-the-fly. We consider Aalta version 2 [51] and version 1 [50], that implement different search algorithms for building the automata.

PLTL is a traditional solver [68] that implements three different algorithms in each version. The latest version is a symbolic approach based on BDDs [34, 58], while the two previous versions are based on two different algorithms for computing the Tableau [69, 78].

Table 2: Solvers and versions used.

Solver	Releases	Algorithms	Configs.
NuSMV	2.6.0, 2.5.4, 2.4.3	BMC, BDD	6
Black	0.9.2, 0.7.4, 0.5.2	Tableau	10
Aalta	v2, v1	Automata, Tableau	2
PLTL	-	BDD, Tableau (2 variants)	3

4.4 Setting and Evaluation

SpecBCFuzz is implemented in Java using the JMetal framework [62], that instantiates the NSGA-III algorithm and integrates all the solvers we test. It also uses the OwL library [46] to parse and manipulate the LTL formulas. We make our tool, seeded formulas and divergences, as well as the bug-triggering formulas, publicly available at: **https://github.com/SpecBCFuzz/repo**. We configure several parameters of the NSGA-III algorithm based on the findings of other studies and exploratory analyses. Particularly, at every generation we preserve a population of 100 individuals, the mutation operator is always applied to selected individuals, while the crossover is applied with a probability of 0.1. Moreover, for computing the semantic similarity objective, we set a timeout of 30 seconds for the model counting approach.

We run the fuzzing campaigns for 48 hours or until $1,000$ individuals ($\mathcal{S}'$) were generated, whichever happened first, with no re-starts. Notice that, assuming we have, for instance, 10 boundary conditions in Δ, the fuzzing campaign will produce $1,000 \times 10$ input formulas to test the 21 solvers' versions, i.e., a total of $210,000$ sat invocations. We set a timeout of 300 seconds for the solver.

To answer RQ2, we run SpecBCFuzz by disabling some of the objectives and to answer RQ3, we fine-tune a (probabilistic) grammar-based fuzzer [79]. The grammar-based fuzzer produces random LTL formulas by randomly visiting non-terminal and terminal nodes. We varied several parameters to produce more diverse LTL formulas: the maximum of literals (ranging from 1 to 9), the maximum number of non-terminals reached (ranging from 2 to 10), terminal choice probability (from 0.20 to 0.44), and boolean constant probability (ranging from 0.05 to 0.29). Moreover, the maximum number of formulas generated is 20,000 (20 times more than with SpecBCFuzz).

We ran SpecBCFuzz and the grammar fuzzer on an HPC cluster. Each node has a Xeon E5 2.4GHz, with 16 CPUs-nodes available and 4GB of memory per CPU. The operating system is Centos Linux, version 7. Since both tools are stochastic, we repeat the experimental process 10 times, to avoid random bias.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Robustness and Failures Patterns

5.1.1 SpecBCFuzz Effectiveness. Table 3 summarizes the number of formulas generated by SpecBCFuzz that trigger a bug in each solver configuration, summarizing to a total of 368,716 unique bug-triggering formulas (since a same formula can reveal different symptoms in other solvers and versions). Noticeably, SpecBCFuzz revealed diverse kinds of faults in 18 out of the 21 studied solver configurations. The only exceptions were all versions of NuSMV BDD, suggesting that this is the most robust solver (according to our experiments).

Overall, crashes and flakiness bugs are the predominant observed symptoms. It is worth to remark that flakiness is essentially a soundness issue too, since it implies the generation of incorrect results, affecting the robustness and correctness of solvers. Regarding performance, we observed very clear issues in cases in which solvers do not produce an output within 24 hours of execution, while other solvers do it very efficiently. This possibly points to hang loops in the implemented algorithms.

Table 3: Number of soundness, crashes, flaky and performance ($\mathcal{B}_s, \mathcal{B}_c, \mathcal{B}_f, \mathcal{B}_p$) bug-triggering formulas generated by SpecBCFuzz.

version	$\#\mathcal{B}_s$	$\#\mathcal{B}_c$	$\#\mathcal{B}_f$	$\#\mathcal{B}_p$
NuSMV				
2.6.0 + BMC	0	44,394	351	0
2.5.4 + BMC	0	49,689	351	0
2.4.3 + BMC	0	49,828	490	0
Black				
0.9.2 + Tableau	0	86	0	72
0.7.4 + Tableau	0	86	0	72
0.5.2 + Tableau	0	303,924	0	72
Aalta				
v2 + Tableau	78	3,570	4,698	3,560
v1 + Automata	78	3,570	4,606	3,560
PLTL				
BDD	75	0	0	0
Tableau graph-based	0	0	0	3
Tableau multipass-based	0	0	0	3
Total: 368,716	**153**	**357,673**	**10,135**	**3,635**

5.1.2 Failures Patterns. Same bug-triggering formula can induce the same or different failure in the solvers. Then, we aim at performing a more in depth analysis by clustering the failures according to the input/outputs relations, i.e., between the bug-triggering formulas and the symptoms shown by the solvers' executions. Table 4 summarizes 16 particular clusters capturing different buggy

symptoms we identified for each solver, and later on we discuss symptoms potentially revealing further performance bugs.

NuSMV. Cluster number 1 includes a set of LTL formulas that trigger a crash ("segmentation fault") in all the versions of NuSMV BMC. The second cluster characterizes a set of LTL formulas that reveals a flaky behavior in BMC versions 2.6.0 and 2.5.4, since from time to time these solver configurations crash when run on a same input formula. We reported these two bugs to the corresponding development team, and they replied that they are currently investigating the issues. Clusters number 3 and 4 characterize crashes and flaky executions (often crashes without error messages) in past BMC versions of the solver (2.5.4 and 2.4.3). Notice that bugs triggered by clusters 3 and 4 have been fixed by developers in the latest version of the solver.

Black. LTL formulas in Cluster 7 are of the form "$((f_1 \mathcal{W} f_2))$", e.g., $((a \mathcal{W} b))$). These formulas trigger a parser failure in Black version 0.5.2, which has been fixed in recent versions. Cluster 5 contains formulas that trigger a crash (with a "killed" message) in all versions of Black. We reported this bug to the developers, and it was confirmed, although it has not yet been fixed. For formulas in Cluster 6, all the versions of Black cannot produce an outcome within 24 hours of execution. SpecBCFuzz oracles classified this symptom as a performance bug, but it can also pinpoint to bugs that lead to a hang loop. We reported this bug and the developers answered that are investigating the issue.

Aalta. SpecBCFuzz generated LTL formulas that trigger all kinds of failures in Aalta (both versions). Cluster 8 includes formulas that induce Aalta to return an incorrect satisfiability answer, i.e., soundness issues. Cluster 9 contains formulas triggering different kinds of crashes. Clusters 10 and 11 capture different sets of formulas that induce flaky behavior in the two versions of Aalta, by producing different satisfiability results from time to time. Clusters 12, 13 and 14 capture different sets of LTL formulas that reveal different symptoms in Aalta v1 and v2, leading to crashes, flakiness, and performance bugs (as cataloged according to SpecBCFuzz's oracles). Again, formulas for which Aalta does not produce an output within 24 hours can potentially pinpoint hang loops in the code. We reported the 6 bugs from clusters 8-9 and 11-14, affecting the latest version (v2) of the solver, to the developers (issues corresponding to cluster 10, affecting v1, were solved in v2). We received the confirmation of 2 out of these 6 bugs (clusters 8 and 11), and developers are investigating the remaining four issues.

PLTL. Cluster 15 contains LTL formulas of the form "$\square\neg(\mathtt{f})$", for which PLTL BDD always produces an incorrect satisfiability answer. We reported this bug to the developers, but we received no response yet. Finally, Cluster 16 contains LTL formulas for which the past version of the solver implementing different Tableau algorithms, cannot produce an outcome within 24 hours of execution, while other solvers answer in seconds. These performance issues were not observed in the latest version of the solver, based on BDD.

Overall, **3 out of the 16** bugs found have been **confirmed** by the developers, **7** are currently **under analysis** by the developers, and **5** were **fixed** in followup versions of the solvers. We have not yet received a response by the corresponding development team for 1 of the reported bugs.

Additionally, we collected 315 clusters of LTL formulas that lead different combinations of solvers to reach the analysis timeout (set in 300 seconds by SpecBCFuzz). Although these clusters may not represent performance bugs, since the solvers can indeed output a satisfiability result in less than 24 hours, they constitute an interesting test-bed, which we classified by version and algorithm implemented, that can challenge the performance of the solvers, and can be used for regression testing purposes. We make these data available in our accompanying website.

5.2 RQ2: Importance of Fitness Objectives

We perform an ablation study to assess SpecBCFuzz' effectiveness with deactivated fitness objectives. In configuration (Sem+#BCs) we deactivate the syntactic similarity objective, making SpecBCFuzz guided by the semantic similarity and the number of boundary conditions that remain unsatisfiable (objective *#BCs*). In configuration (Syn+#BCs) we deactivate the semantic similarity objective, while in configuration (Syn+Sem) #BCs is deactivated, making SpecBCFuzz guided only by the similarity metrics.

Figure 4 summarizes the impact of deactivating, one by one, the fitness objectives, on our results. We see that by deactivating the syntactic or semantic similarities, the number of bugs detected is drastically reduced. Configurations (Syn+#BCs) and (Sem+#BCs) can only detect the Cluster 1 failures (crashes in NuSMV BMC - all versions) and Cluster 7 (syntax parsing failures in Black 0.5.2).

On the other hand, deactivating the #BCs objective in configuration (Syn+Sem) produces a minor, still important, impact in SpecBCFuzz's effectiveness, compared to deactivating the similarity metrics. (Syn+Sem) can trigger faults captured by Clusters 1, 3 and 4 (i.e., crashes and flaky behaviors in NuSMV BMC), 9, 10 and 11 (i.e., crashes and flaky behaviors in Aalta v1 and v2), and 15 (i.e., soundness issues in PLTL BDD with formulas of the form $\square\neg\mathtt{f}$).

Overall, the combination of the three fitness objectives contributes significantly to SpecBCFuzz's effectiveness in revealing more diverse buggy symptoms in LTL solvers.

5.3 RQ3: Validation of SpecBCFuzz's Principles

Figure 5 shows that both principles of our approach are relevant to revealing faults in LTL solvers. We observe that, if no boundary condition is used when feeding the solvers, but we still search in the vicinity ($\{\mathcal{S}\} \cup \{\mathcal{S}'_j\}_j$), we are only able to produce LTL formulas that trigger 2 kinds of failures, captured by Clusters 1 and 7. When instead boundary conditions are directly used, but no search is performed ($\{\mathcal{S} \wedge \delta_i\}_i$), we only find faults corresponding to Clusters 1 and 7.

It is worth to highlight that **the large set of LTL benchmark formulas** did not reveal any fault. We then developed a probabilistic grammar-based fuzzer, by carefully setting probabilities and parameters to the grammar's production rules. Since this approach explores a broad spectrum of the search (20,000 formulas – 20 times more than SpecBCFuzz), it can produce LTL formulas that trigger faults similar to 11 Clusters from Table 4. However, we observe that for four clusters, the grammar-based fuzzer only manages to trigger such faults very rarely (3, 4, 5 and 7 times, respectively, out of the 10 runs we performed). Moreover, we also observe that for most of the clusters, this approach produces very few bug-triggering

Table 4: Cluster of the observed symptoms.

#Cluster	*Oracle Type*	Symptoms Observed	Versions
		NuSMV	
1	Crash	Crash and Exception thrown with message: *Segmentation fault (core dumped)*	BMC - all versions
2	Flaky and Crash	From time to time it crashes and throws the message: *Segmentation fault (core dumped)*	BMC - 2.6.0, 2.5.4
3	Crash	Crash and Exception thrown with message: *Segmentation fault (core dumped)*	BMC - 2.5.4, 2.4.3
4	Flaky and Crash	From time to time it crashes without error message.	BMC - 2.4.3
		Black	
5	Crash	Crash and Exception thrown with message: *Killed*	all versions
6	Performance	Black does not respond in 24 hours (possible hang loop)	all versions
7	Crash	Syntax parser failure with formulas of the form "$((f_1 \mathcal{W} f_2))$".	0.5.2
		Aalta	
8	Soundness	Aalta returns an incorrect satisfiability answer	all versions
9	Crash	Crash and Exception thrown with message: *Assertion failed.*	v1
	Crash	Crash without error message.	v2
10	Flaky	Altaa produces different output from time to time.	v1
11	Flaky	Altaa produces different output from time to time.	v2
12	Crash	Version 1 crashes with the message: *assertion failed: qi check*	v1
	Performance	Version 2 does not respond in 24 hours (possible hang loop)	v2
13	Crash	Version 1 crashes with the message: *Assertion failed*	v1
	and Flaky	Version 2 crashes from time to time without message	v2
14	Crash	Version 1 crashes with the message: *assertion failed: clsnum check*	v1
	Performance	Version 2 does not respond in 24 hours (possible hang loop)	v2
		PLTL	
15	Soundness	Formulas of the form "□¬(formula)" always return an incorrect satisfiability answer.	BDD
16	Performance	Both Tableau versions do not answer in 24 hours (possible hang loop)	Tableau

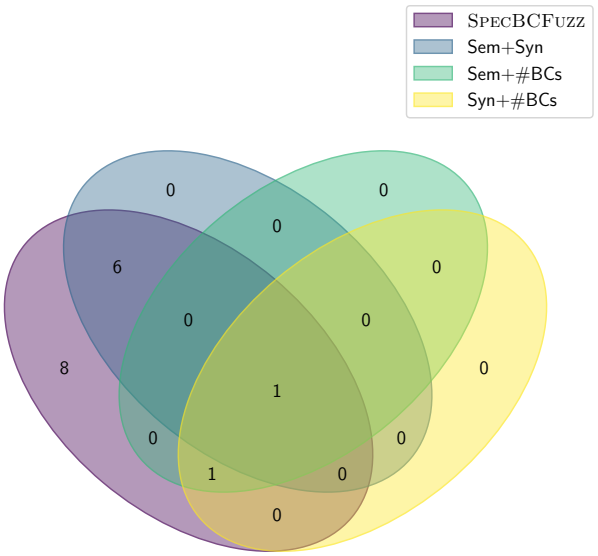

Figure 4: Ablation Study.

formulas in relation to the numbers produced by SpecBCFuzz (10X to 100X fewer), except for the bugs in PLTL and the syntax parser crash in Black, for which it produces a similar amount of triggering formulas. Surprisingly, our fine-tuned grammar-based fuzzer can trigger a crash in PLTL BDD, with the error message "Segmentation fault (core dumped)", not triggered by SpecBCFuzz (i.e., the symptoms observed are not covered by the patterns in Table 4). Overall, while it shows a better performance than SpecBCFuzz for testing the PLTL solver, it achieves a limited performance when testing all the other solvers.

To conclude, our results suggest that *the combination of LTL specifications with boundary conditions and vicinity search is an effective heuristic for producing bug-triggering LTL formulas.*

6 THREATS TO VALIDITY

Threats to external validity concern the solvers and seeded formulas we used in our evaluation. We searched for LTL solvers supporting different heuristics, and we gathered formulas with divergences from the literature that are typically used for the evaluation of requirements analysis tools. Results may not generalize to other solvers, logical languages and domains (for instance, when a language other than LTL is employed). To mitigate this threat, we also studied the application of SpecBCFuzz to other kinds of solvers that take LTL formulas as input, such as model checkers. We included Spin [41] (version 6.5.1) as one of the subjects under analysis, and SpecBCFuzz did not detect any fault.

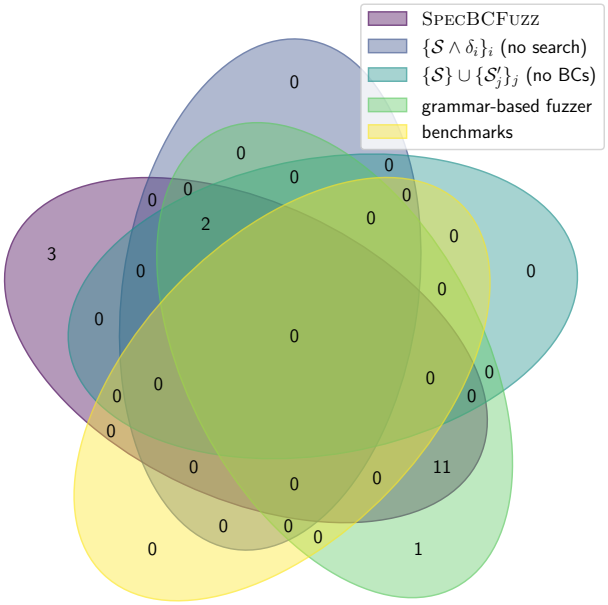

Figure 5: Divergences and Vicinity Exploration Matters

Threats to internal validity relate to the implementation details. To mitigate this risk, we rely on reliable third-party frameworks and libraries such as the NSGA-III implementation of the JMetal framework [62]. Since SpecBCFuzz and the baselines are stochastic, we repeat our experiments 10 times, following research guidelines for how such algorithms should be evaluated [3].

Our assessment metrics, namely the number of generated formulas that trigger a fault in a solver, is intuitive and reflect the effectiveness of SpecBCFuzz. Since our analysis is black-box, identified bug-triggering formulas may relate to different bugs in the solver to fix. We then performed a follow-up analysis to cluster and classify the sets of formulas showing different faulty symptoms. Related to our differential oracles, in addition to the re-runs, we performed extensive manual analyses to double-check the outcomes and confirm the identified faults. For correctness bugs, we applied a delta-debugging like approach to obtain simpler sat-preserving formulas, that confirm the original wrong satisfiability outcomes. Crashes were confirmed by a simple reproduction, while flakiness bugs were confirmed by an extensive re-run of the likely flaky solvers (in our evaluation, all flaky behaviors were reproduced and confirmed). Performance bugs were detected by adopting a very conservative threshold (24 hours, or 300 times slower behavior compared with other solvers).

7 RELATED WORK

Fuzzing has been used for testing solvers. Brummayer et al. [14] presented a black-box grammar-based fuzzing for propositional (SAT) and quatifier-free (QBF) solvers. Among the various fuzzers to test solvers, CNFuzz generates random CNF formulas based on a given CNF grammar and set of parameters, e.g., maximum layers, width, and variables. FuzzSAT generates random formulas in the form of random boolean circuit (RBC), a kind of directed acyclic graphs. QBFuzz works similarly, but for producing QBF formulas. The empirical evaluation shows their corresponding capabilities for finding bugs in SAT and QBF solvers.

SMT solvers have been the target of several testing approaches. StringFuzz implements string generators and transformers to increase the complexity of string constraints [11], by replacing literals and operators and swapping non-leaf nodes with leaf nodes. StringFuzz also can include a seed-based strategy, fed with realistic regular expression, to improve the fuzzing testing. Bugariu and Muller aimed at generating more complex string operations [15], that are sat/unsat by construction, which are used as test oracles. Other approaches, like STORM, also adopt seed-based and mutation fuzzing techniques [57]. SpecBCFuzz also implements string transformations, i.e., mutation and crossover operators, to evolve LTL formulas. It uses requirements specifications with divergences as seeds to improve fault detection of LTL solvers. Other works have used evolutionary algorithms to produce LTL formulas with different goals, e.g., for specification repair [13, 17], and relevant to this paper, to identify boundary conditions [28].

The recently developed HistFuzz [72] proposes to use seed-based buggy skeletons, crafted from historical bug reports. Furthermore, DIVER [44] takes an original satisfiable formula, for which a model is built, applies unrestricted mutations during the search, and uses the model as an oracle (if the mutated formula is consistent with the model, then it should be satisfiable). These approaches were found effective for finding bugs in popular SMT solvers as Z3 [25], CVC5 [4], and dReal [35]. SpecBCFuzz is also based on semantic properties and assumes that LTL specifications with divergences, i.e., conflicting specifications in the context of goal-oriented requirements, are likely to trigger faults in LTL solvers.

Schuppan and Darmawan [68] conducted an empirical study to assess the performance of LTL solvers, based on a large benchmark of 3,723 LTL formulas of different complexities (that we used in RQ3) to challenge the solvers. To conclude, our work is the first approach that fuzzes LTL solvers and reveals crashes, correctness bugs, flakiness issues, and performance bugs. SpecBCFuzz was shown to be effective in triggering different kinds of bugs in the latest and past versions of very efficient LTL solvers.

8 CONCLUSION

We presented SpecBCFuzz, a fuzzing method that combines boundary conditions and vicinity exploration for testing LTL solvers. We showed that SpecBCFuzz is effective in producing formulas (368,716) that trigger different kinds of bugs (soundness issues, crashes, flakiness and performance issues) in 18 out of the 21 studied solvers' configurations. In our evaluation, SpecBCFuzz did not trigger any bug in the 3 versions of NuSMV BDD, suggesting that this is currently the most robust LTL solver.

9 ACKNOWLEDGMENTS

This work is supported by the Luxembourg National Research Funds (FNR) through the CORE project grant C19/IS/13646587/RASoRS. Nazareno Aguirre is also supported by ANPCyT PICTs 2019-2050 and 2020-2896, an Amazon Research Award, and by EU's Marie Sklodowska-Curie grant No. 101008233 (MISSION).

REFERENCES

[1] 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.

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[3] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 1–10, New York, NY, USA, 2011. Association for Computing Machinery.

[4] Haniel et al. Barbosa. cvc5: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.

[5] Dirk Beyer and Marie-Christine Jakobs. Coveritest: Cooperative verifier-based testing. In Reiner Hähnle and Wil M. P. van der Aalst, editors, *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11424 of *Lecture Notes in Computer Science*, pages 389–408. Springer, 2019.

[6] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.

[7] Dirk Beyer and Stefan Löwe. Explicit-state software model checking based on CEGAR and interpolation. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7793 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2013.

[8] 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, page 193–207, Berlin, Heidelberg, 1999. Springer-Verlag.

[9] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. Hotfuzz: Discovering temporal and spatial denial-of-service vulnerabilities through guided micro-fuzzing. *ACM Trans. Priv. Secur.*, 25(4), jul 2022.

[10] Roderick Bloem, Robert Könighofer, and Martina Seidl. Sat-based synthesis methods for safety specs. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 1–20, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[11] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. Stringfuzz: A fuzzer for string solvers. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 45–51, Cham, 2018. Springer International Publishing.

[12] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Matías Brizzio, Maxime Cordy, Mike Papadakis, César Sánchez, Nazareno Aguirre, and Renzo Degiovanni. Automated repair of unrealisable ltl specifications guided by model counting. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '23, page 1499–1507, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of sat and qbf solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010*, pages 44–57, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[15] Alexandra Bugariu and Peter Müller. Automatically testing string solvers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1459–1470, New York, NY, USA, 2020. Association for Computing Machinery.

[16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2019.

[17] Luiz Carvalho, Renzo Degiovanni, Matías Brizzio, Maxime Cordy, Nazareno Aguirre, Yves Le Traon, and Mike Papadakis. Acore: Automated goal-conflict resolution. In Leen Lambers and Sebastián Uchitel, editors, *Fundamental Approaches to Software Engineering*, pages 3–25, Cham, 2023. Springer Nature Switzerland.

[18] Konstantin Chukharev, Dmitrii Suvorov, Daniil Chivilikhin, and Valeriy Vyatkin. Sat-based counterexample-guided inductive synthesis of distributed controllers. *IEEE Access*, 8:207485–207498, 2020.

[19] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, pages 359–364, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[20] Alessandro Cimatti, Enrico Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Integrating bdd-based and sat-based symbolic model checking. In Alessandro Armando, editor, *Frontiers of Combining Systems*, pages 49–56, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[21] Alessandro Cimatti, Enrico Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Integrating bdd-based and sat-based symbolic model checking. In *Proceedings of the 4th International Workshop on Frontiers of Combining Systems*, FroCoS '02, page 49–56, Berlin, Heidelberg, 2002. Springer-Verlag.

[22] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[23] E. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. *Another look at LTL model checking*. 1997.

[24] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19:7–34, 01 2001.

[25] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[26] Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, 2014.

[27] 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.

[28] 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.

[29] 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.

[30] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[31] Maolin Sun et al. Validating smt solvers via skeleton enumeration empowered by historical bug-triggering inputs. In *ICSE*, 2023.

[32] 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.

[33] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, WOOT'20, USA, 2020. USENIX Association.

[34] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[35] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dreal: An smt solver for nonlinear theories over the reals. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24*, pages 208–214, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[36] Luca Geatti, Nicola Gigante, and Angelo Montanari. BLACK: A fast, flexible and reliable LTL satisfiability checker. In Dario Della Monica, Gian Luca Pozzato, and Enrico Scala, editors, *Proceedings of the 3rd Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis hosted by the Twelfth International Symposium on Games, Automata, Logics, and Formal Verification (GandALF 2021), Padua, Italy, September 22, 2021*, volume 2987 of *CEUR Workshop Proceedings*, pages 7–12. CEUR-WS.org, 2021.

[37] Luca Geatti, Nicola Gigante, Angelo Montanari, and Gabriele Venturato. Past matters: Supporting ltl+past in the BLACK satisfiability checker. In Carlo Combi, Johann Eder, and Mark Reynolds, editors, *28th International Symposium on Temporal Representation and Reasoning, TIME 2021, September 27-29, 2021, Klagenfurt, Austria*, volume 206 of *LIPIcs*, pages 8:1–8:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[38] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 412–416, 2001.

[39] Christopher Hahn, Frederik Schmitt, Jens U. Kreber, Markus Norman Rabe, and Bernd Finkbeiner. Teaching temporal logics to neural networks. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria,*

May 3-7, 2021. OpenReview.net, 2021.

[40] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 230–243, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[42] Hakjoo Oh Jongwook Kim, Sunbeom So. Diver: Oracle-guided smt solver testing with unrestricted random mutations. In *ICSE*, 2023.

[43] Hong Jin Kang and David Lo. Adversarial specification mining. *ACM Trans. Softw. Eng. Methodol.*, 30(2):16:1–16:40, 2021.

[44] Jongwook Kim, Sunbeom So, and Hakjoo Oh. Diver: Oracle-guided smt solver testing with unrestricted random mutations. In *Proceedings of the ACM/IEEE 45th International Conference on Software Engineering*, ICSE '23, 2023.

[45] J. Kramer, J. Magee, M. Sloman, and A. Lister. CONIC: an integrated approach to distributed computer control systems. *Computers and Digital Techniques, IEE Proceedings E*, 130(1):1+, 1983.

[46] Jan Kretínský, Tobias Meggendorfer, and Salomon Sickert. Owl: A library for ω-words, automata, and LTL. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 543–550. Springer, 2018.

[47] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, mar 2000.

[48] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[49] 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.

[50] Jianwen Li, Lijun Zhang, Geguang Pu, Moshe Y. Vardi, and Jifeng He. Ltl satisfiability checking revisited. In *2013 20th International Symposium on Temporal Representation and Reasoning*, pages 91–98, 2013.

[51] Jianwen Li, Shufang Zhu, Geguang Pu, and Moshe Y. Vardi. Sat-based explicit ltl reasoning. In Nir Piterman, editor, *Hardware and Software: Verification and Testing*, pages 209–224, Cham, 2015. Springer International Publishing.

[52] Miqing Li and Xin Yao. Quality evaluation of solution sets in multiobjective optimisation: A survey. *ACM Comput. Surv.*, 52(2), mar 2019.

[53] Weilin Luo, Hai Wan, Jianfeng Du, Xiaoda Li, Yuze Fu, Rongzhen Ye, and Delong Zhang. Teaching ltlf satisfiability checking to neural networks. In Lud De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 3292–3298. International Joint Conferences on Artificial Intelligence Organization, 7 2022. Main Track.

[54] Weilin Luo, Hai Wan, Xiaotong Song, Binhao Yang, Hongzhen Zhong, and Yin Chen. How to identify boundary conditions with contrasty metric? In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1473–1484. IEEE, 2021.

[55] Weilin Luo, Hai Wan, Delong Zhang, Jianfeng Du, and Hengdi Su. Checking ltl satisfiability via end-to-end learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

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

[57] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in smt solvers using blackbox mutational fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 701–712, New York, NY, USA, 2020. Association for Computing Machinery.

[58] Will Marrero. Using bdds to decide ctl. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 222–236, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[59] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. Linear-time temporal logic guided greybox fuzzing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1343–1355. ACM, 2022.

[60] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, dec 1990.

[61] Roberto Natella and Van-Thuan Pham. Profuzzbench: A benchmark for stateful protocol fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[62] Antonio J. Nebro, Juan J. Durillo, and Matthieu Vergne. Redesigning the jmetal multi-objective optimization framework. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO Companion '15, page 1093–1100, New York, NY, USA, 2015. Association for Computing Machinery.

[63] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 398–401, New York, NY, USA, 2019. Association for Computing Machinery.

[64] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 329–340, New York, NY, USA, 2019. Association for Computing Machinery.

[65] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[66] Mark Reynolds. A new rule for ltl tableaux. In *International Symposium on Games, Automata, Logics and Formal Verification*, 2016.

[67] Kristin Y. Rozier and Moshe Y. Vardi. LTL satisfiability checking. *STTT*, 12(2):123–137, 2010.

[68] Viktor Schuppan and Luthfi Darmawan. Evaluating ltl satisfiability solvers. In *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis*, ATVA'11, page 397–413, Berlin, Heidelberg, 2011. Springer-Verlag.

[69] Stefan Schwendimann. A new one-pass tableau calculus for pltl. In Harrie de Swart, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 277–291, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[70] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending sat solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 244–257, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[71] Dominic Steinhöfel and Andreas Zeller. Input invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 583–594, New York, NY, USA, 2022. Association for Computing Machinery.

[72] Maolin Sun, Yibiao Yang, Ming Wen, Yongcong Wang, Yuming Zhou, and Hai Jin. Diver: Oracle-guided smt solver testing with unrestricted random mutations. In *Proceedings of the ACM/IEEE 45th International Conference on Software Engineering*, ICSE '23, 2023.

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

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

[75] Moshe Vardi. An automata-theoretic approach to linear temporal logic. *Logics for Concurrency*, pages 238–266, 1996.

[76] Willem Visser, Klaus Havelund, Guillaume P. Brat, and Seungjoon Park. Model checking programs. In *The Fifteenth IEEE International Conference on Automated Software Engineering, ASE 2000, Grenoble, France, September 11-15, 2000*, pages 3–12. IEEE Computer Society, 2000.

[77] Matt Walker, Parssa Khazra, Anto Nanah Ji, Hongru Wang, and Franck van Breugel. Jpf-logic: A framework for checking temporal logic properties of java code. *SIGSOFT Softw. Eng. Notes*, 48(1):32–36, jan 2023.

[78] Pierre Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, 28(110/111):119–136, 1985.

[79] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Probabilistic grammar fuzzing. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2023. Retrieved 2023-01-07 15:01:16+01:00.

[80] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Comput. Surv.*, 54(11s), sep 2022.