UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2025-015
The Faculty of Sciences, Technology and Medecine

Dissertation

Defence held on January 10th, 2025 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Luiz Matheus de Alencar Carvalho
Born on 23th September 1994 in Aracaju (Brazil)

Enhancing LTL Specifications and Solver Reliability through Search Techniques

Dissertation defence committee

Abstract

Linear temporal logic (LTL) is a powerful logical formalism that includes descriptive operators for future events that deal with concurrent behaviors. LTL has a general set of operators that impose high expressiveness and complexity. The formalism includes safety, liveness, and fairness properties. LTL is commonly used as a specification language for distributed, concurrent, and hardware systems. Moreover, LTL is a common formalism in Goal-Oriented Requirements Engineering (GORE). The requirement methodology has taken advantage of the analysis techniques associated with LTL. In this manner, we present ACoRe, an automated approach to resolve goal-conflicts in requirements specifications expressed in LTL. ACoRe explores syntactic modifications of the specifications to remove all the previously identified conflicts while preserving consistency. Moreover, ACoRe also computes the syntactic and semantic similarity, in which the syntactic similarity refers to the distance between the text representations of the original specification and the candidate repair, while the semantic similarity refers to the behavior of the system shared by the original specification and the candidate repair. We evaluated ACoRe on formal specifications collected from the literature and benchmarks. In summary, the empirical results present a reasonable number of repairs in the Pareto-optimal set. Also, we show that ACoRe obtains better results when genetic algorithms are used, as opposed to other search algorithms, measured in terms of quality indicators, while at the same time, it produces more repairs that do not introduce new conflicts.

On the basis of the high level of expressiveness provided by LTL, tools have been created to mine LTL properties based on trace artifacts. A trace describes the relevant events that occur or the behavior changes of concurrent systems. The LTL mining tools analyze patterns or constraints in these traces and propose LTL properties that are valid in the provided traces. However, there is a gap in the evaluation of the capability of mining tools. For instance, if the tools are able to propose correct properties or reach a soundness set of LTL properties. Therefore, we propose an evaluation of LTL mining tools. First, we assess them based on the capability to find a ground truth set of LTL specifications. Second, we observe the capability to cover the ground truth set of LTL specifications. Our

evaluation shows that the mining tools do not find LTL properties in a ground truth composed of formal specifications. Moreover, the experiment also indicates that the mining tools have low precision in the semantic coverage of the ground truth. The empirical results suggest that enriching or completing existing specifications by mining additional LTL traces is still an open research area, and the current tools are useless for improving existing specifications.

Furthermore, LTL solvers have been proposed to address the LTL satisfiability problem. In an effort to answer the satisfiability problem, the LTL solvers were designed around several algorithms. Among the additional efforts made, developments have also been made in terms of improving performance gains. Among the applications are complex contexts such as program analysis, software verification, symbolic execution, program synthesis, model checking, and artificial intelligence reasoning. These application domains generally require high-quality standards.

Our work also presents a new fuzzing technique for LTL solvers based on a search-based approach called SpecBCFuzz. Among the fitness functions, the boundary conditions are extensively evaluated during the search process, which progressively resolves the boundary conditions in the LTL specifications that were provided as seeds. Thus, the evolved specifications are valid repairs for the previously identified diverges. We claim that the search process for resolving boundary conditions efficiently produces failure-inducing input. Moreover, additional fitness, such as semantic and syntactic similarities, can constrain the search space and maintain the seeds given by the specifications as representative of realistic LTL specifications.

Our empirical evaluation relies on differential fuzzing testing and inconsistency patterns. The result suggests that SpecBCFuzz is significantly more efficient than a baseline approach (e.g., LTL probabilistic grammar fuzzing). In addition, the empirical study points out a more significant size effect of the inconsistency patterns identified during the execution of the empirical study. Furthermore, our empirical evaluation presents 16 bugs in the evaluated LTL solvers, detecting bugs in 18 out of the 21 solvers' configurations we study. In addition, we present performance warning patterns. The set of warning patterns has the potential to indicate improvements in a particular solver for releases, heuristics set, or theoretical guidelines for a specific type of LTL satisfiability algorithm.

Moreover, we modify portfolio decisions to incorporate a confirmation answer. In other words, the portfolio does not provide the fastest answer. The portfolio replicates the answer when at least two solvers agree on a given input. It was formulated based on our previous study that identified bugs in LTL solvers and demonstrated that a considerable number of these bugs are not inherent to a single input across different solvers. Our empirical results present bugs in the pure portfolio, whereas a search-based fuzzing approach does not find a single bug for the reliable portfolio. Furthermore, we also evaluated the reliable portfolio in terms of performance. The empirical results present the lowest runtime for the reliable portfolio compared to a single robust LTL solver (NuSMV). In summary, a reliable portfolio is capable of increasing soundness in LTL solvers and is also associated with a gain in performance compared to an individual LTL solver that contains a high level of soundness.

Acknowledgments

I hope this message finds you well. I am writing with a heart full of gratitude to express my sincerest thanks.

First, I express my gratitude to all of the professors who have provided me with intellectual support and encouragement and who have been instrumental in making this work and previous works possible. I also need to express my gratitude to my family. They were essential for my personal development.

In addition, I express my appreciation to my wife Silvia Rodrigues. I am indebted to my wife for her unwavering understanding and patience throughout my academic journey. Beyond the tangible impact, I want to highlight the emotional and personal significance of your support. These feelings are treasures in themselves, and I am lucky to have someone like you in my life.

Furthermore, I am grateful for the support that allowed me to maintain a strong passion for learning. I am also grateful to my friends and colleagues, who provided a nurturing environment for group study, encouragement, and camaraderie. Such moments remind me of the importance of connection and community.

Luiz Carvalho
Luxembourg
December 2024

Contents

1 Introduction **1**
1.1 Context 2
1.1.1 Formal Methods 2
1.1.2 Formal Specifications 3
1.1.3 Linear Temporal Logic 4
1.2 Challenges 5
1.2.1 Automated Specification Repair 5
1.2.2 LTL Specification Mining 5
1.2.3 Soundness of LTL Solvers 6
1.2.4 Reliable Portfolio of LTL Solvers 6
1.3 Overview of the Contributions 7
1.4 Organization of the Dissertation 9

2 Related Work **11**
2.1 Automated LTL Specification Repair 12
2.2 LTL Specification Mining 14
2.3 Soundness of LTL Solvers 16
2.4 Reliable Portfolio of LTL Solvers 17

3 Automated Goal-Conflict Resolution **21**
3.1 Introduction 23
3.2 Background 25
3.2.1 Linear-Time Temporal Logic 25
3.2.2 Model Counting 26
3.2.3 Multi-Objective Optimization Algorithms 27
3.3 Goal-oriented Specifications and Conflicts 29
3.3.1 The Goal-Conflict Resolution Problem 30
3.3.2 Motivating Examples 31
3.4 ACoRe: Automated Goal-Conflict Resolution 35
3.4.1 Search Space 36

3.4.2 Initial Population 36
3.4.3 Search Objectives 36
3.4.4 Evolutionary Operators 38
3.5 Research Questions 41
3.6 Experimental Setup 41
3.6.1 Subjects 41
3.6.2 Experimental Procedure 42
3.6.3 Quantitative Evaluation Metrics 44
3.6.4 Implementation 44
3.7 Experimental Results 45
3.7.1 RQ1: Effectiveness of ACoRe 45
3.7.2 RQ2: Comparison with the Ground-truth 46
3.7.3 RQ3: New conflicts introduced by repairs 47
3.7.4 RQ4: Comparing the four Multi-objective Optimization Algorithms 49
3.8 Threat to Validity 51
3.9 Conclusion 52

4 Evaluating Specification Mining in Linear Temporal Logic 55
4.1 Introduction 57
4.2 Background 60
4.2.1 Linear Temporal Logic 60
4.2.2 Mining Problem 61
4.2.3 Linear Temporal Logic to Automaton 61
4.3 Pattern Match 62
4.4 Constraint Solver 63
4.5 Experiment 64
4.6 Research Questions 65
4.6.1 Experimental Instruments 66
4.6.2 Experimental Setting 67
4.7 Evaluation 67
4.7.1 Effective 67
4.7.2 Capability 69
4.8 Threat to Validity 70
4.9 Conclusion 71

5 Fuzzing Linear Temporal Logic Solvers 73
5.1 Introduction 74
5.2 Background 76
5.2.1 Linear Temporal Logic (LTL) 77
5.2.2 Solvers 78

5.2.3 Fuzzing 79
5.3 Boundary Conditions 80
5.4 SpecBCFuzz 82
5.4.1 Evoluationary Search 83
5.4.2 Search Based Fuzzing 84
5.4.3 Fitness 85
5.4.4 Evolutionary Operators and Selection 86
5.4.5 Differential Fuzzing 88
5.5 Empirical Evaluation 89
5.5.1 LTL Solvers 90
5.5.2 Seeds 92
5.5.3 Fuzzing and Experimental Settings 93
5.6 Evaluation 94
5.6.1 Bugs and Inconsistency Patterns 94
5.6.2 Performance Issues and Warning Patterns 97
5.6.3 Seeds and Boundary Conditions Matters 99
5.6.4 Statistical Analysis of SpecBCFuzz 100
5.7 Threats to Validity 101
5.8 Conclusion 102

6 Fuzzing Linear Temporal Logic Portfolio **105**
6.1 Introduction 107
6.2 Background 109
6.2.1 Linear Temporal Logic 109
6.2.2 Satisfiability 110
6.2.3 Fuzzing 110
6.3 Portfolio 112
6.3.1 Algorithm Portfolio 112
6.3.2 Algorithm Selection Problem 113
6.3.3 Linear Temporal Logic Portfolio 113
6.3.4 Reliable portfolio 115
6.4 Fuzzing 116
6.5 Families of LTL Benchmarks 117
6.6 Empirical Study 118
6.6.1 Experimental Instruments 119
6.6.2 Experimental Settings 120
6.7 Results 121
6.7.1 Pure Portfolio 121
6.7.2 Reliable Portfolio 122
6.7.3 Runtime Performance 122
6.8 Threats to Validity 124

6.9 Conclusion 125

7 Conclusion **127**

7.1 Summary of Contributions 128

7.2 Perspectives 130

List of publications and tools **i**

List of figures **iii**

List of tables **v**

Bibliography **ix**

1 Introduction

This chapter discusses the context, challenges addressed, and overall contributions of this dissertation. First, we introduce formal methods and specifications. We then introduce Linear Temporal Logic (LTL), a key logical formalism in the context of reactive and concurrent systems. After that, we examine the common challenges in the specification, reasoning, and soundness of LTL. Finally, we summarize the contributions of the dissertation.

Contents

1.1 Context 2
1.2 Challenges 5
1.3 Overview of the Contributions 7
1.4 Organization of the Dissertation 9

1.1 Context

In this section, we introduce relevant and cross-cutting concepts in this dissertation. Among them, we introduce relevant notions of formal methods such as their syntax, semantic, and method. Moreover, we present concepts such as formal specification and linear temporal logic. Subsequently, we also introduce common applications and solid results.

1.1.1 Formal Methods

Formal methods aim to ensure that software meets its requirements. For this objective, formal methods support a precise software description and support rigorous or mathematical analyses. The application domain contains safety-critical software and complex software. That is, the software depends on the correctness or the high level of complexity should be rigorously handled.

Formal methods reside in three components: syntax, semantics, and method. The syntax is composed of a formal language that can be defined by a very well-formed grammar. In terms of semantics, it is composed of a formal semantic that defines the meaning of the objects by mapping them into mathematical definitions or structures. Finally, the method describes algorithms that allow operators under the mentioned objects in practice. In summary, the term formal methods describes a formal specification language (syntax and precise semantic) and a method commonly used for the design [RCS$^+$22].

Several techniques are associated with formal methods, they are used in wide scenarios for verification and validation. Among them, we can mention model checking, theorem proving, runtime verification, and formal testing. In essence, model checking is a tool for verifying models. The model describes the behavior of the software, while correctness properties express the requirement of the software-to-be. In this manner, the model checking provides a formal verification that presents the correctness properties hold for the model. Otherwise, the model checking presents a counterexample that stops the satisfaction of correctness properties [CHV$^+$18; Ben08].

Theorem proving is systems understanding as based on logical principles and is equipped with user-assistive tools that facilitate the construction of proofs. Moreover, they possess the ability to determine the veracity of these proofs in a reliable manner [MB03]. Runtime verification also relies on classical exhaustive verification techniques, such as model checking and theorem proving. Runtime verification aims to analyze finite traces based on events with a view to abstracting a particular observation about a system under analysis [BFF$^+$18].

Regarding formal testing, the model describes the expected behavior in a formal language that follows the mentioned characteristics of well-defined syntax and semantics. Subsequently, a test generator algorithm is expected based on a strong definition of the system under testing, and proof of correctness that the generated tests are sound and exhaustive [Tre08]. However, formal testing is a software testing technique, then it is not able to show the absence of bugs such as all testing techniques, formal testing is able to show evidence of bug presence.

All the mentioned techniques are standard methods and increase the confidence in the correctness of the software under construction. Moreover, formal methods can be applied to improve requirements, detect error, and reduce error introductions. In general, based on formal specification.

1.1.2 Formal Specifications

Formal specification is a set of property definitions of a software-to-be that should satisfy them. It should be done in formal language (e.g., very well-formed statements and supporting reasoning). The formal specification should also consider the real system that covers, the level of abstraction being considered, and the engine for reasoning.

In general, some characteristics are expected by a formal specification. The formal specification must be adequate, internally consistent, unambiguous, complete, satisfied, and minimal [Lam00; BKL08]. That is, the formal specification must fit the given problem and the environment constraints. Besides that, the interpretation of their properties must have a semantic interpretation significantly that turns true all the properties at the same time. Moreover, the formal specification may not have many interpretations and must also be satisfied by a lower-level implementation. Finally, it should not state properties that are irrelevant to the problem.

Furthermore, a formal specification builds capacity associated with the specification. For example, a formal specification is subject to a process of reasoning or proof. Reasoning or proofs are built with a rigorous engine, frequently called mathematical reasoning, and conducted with a mathematical engine. The main goal of reasoning is to identify inconsistencies and demonstrate their existence or, in the last case, the complete absence. For instance, the formal verification performed by model checking tools. The proofs are the refutation or demonstration of conjectures on the basis of formal specification and their following logically from it [Nis99].

Therefore, formal specification brings the ability to a verifiable design process. The requirement engineer may verify with the addition of new features that the design still maintains the previously specified properties and expected behavior. If the process is continued, that is, by individual steps, the requirement engineer may claim the correctness by the construction method. This is also commonly called the design aid.

1.1.3 Linear Temporal Logic

Linear Temporal Logic (LTL) is a logical formalism among the various formal languages that drive formal specifications. It was initially proposed by Pnueli [Pnu77] in 1977 to describe concurrent programs. Moreover, LTL was proposed to guarantee reasoning approaches about concurrent programs. From then on, many applications incorporate LTL in software engineering and co-related fields such as artificial intelligence and image processing to describe temporal, concurrent, and parallel behavior [CTM+17]. In the context of software engineering, LTL is widely used to specify reactive systems [MP92]. That is, systems that have to react to an environment that cannot wait [HP85], such as databases, control systems, and interactive systems.

Furthermore, LTL solvers have been built to address or improve model checking tools, when they conduct a search to answer whether it satisfies the desired property or propose a counterexample that turns it not possible. Historically, the initial work on model checking started in the early 1980s and, at its beginning, combined state exploration and temporal logic to describe properties [CE82].

In general, model-checking tools frequently use this formalism for the formal verification of safety-critical systems such as communications and financial systems [Roz11]. Among the model checking tools, we may state TLA+, Spin, and NuSMV. That contains LTL and additional specification or model languages for distributed and concurrency systems [Lam02; CCG+02; Hol97]. Additional applications of the intersection of model checking and LTL include hardware verification [Gup93].

Moreover, Goal-Oriented Requirements Engineering (GORE) methodologies have also adopted LTL to formally express requirements and have taken advantage of the powerful manual or automatic analysis techniques associated with LTL [vLam09]. The main goal is to improve the quality of the specifications. The analysis phase deals with conflicts through three stages: (1) the identification stage, which identifies conflicts between goals, (2) the assessment stage, regarding the assessing and prioritizing the identified conflicts, and (3) the resolution stage, which the original specification is in fact modified to provide appropriate transformation in the goal model.

Regarding the mentioned formalism, we consider a general set of LTL operators that impose a high level of expressiveness and complexity. The operators describe future events, which are convenient to specify concurrent and reactive systems. Thus, the formalism includes safety, liveness, and fairness properties [OL82]. Additionally, expressiveness may be bounded by fragments as GR(1) [PPS06] or well-known specification patterns such as proposed by Dwyer *et al* [DAC99].

In this manner, LTL solvers are built on the basis of different and complex algorithms. The state-of-the-art presents performance-oriented benchmarks and methods to evaluate LTL solvers. Indeed, LTL solvers are employed in the development of critical systems, and it is of paramount importance that they are reliable. For instance, solvers are utilized as part of verification tools, and erroneous solver results can result in a significant portion of the program under analysis remaining unverified, which in turn increases the risk of failing to identify potential bugs and vulnerabilities.

1.2 Challenges

We identified four main challenges in assessing and improving formal requirements in LTL and their solvers that have not been properly addressed.

1.2.1 Automated Specification Repair

In the context of specification repair, a limited number of automated techniques have been proposed for consideration. Despite the prevalence of automated program repair techniques. Among the limited cases of automated specification repair, Wang *et al.* [WSK19] introduced ARepair, an automated tool to repair a faulty model formally specified in Alloy [Jac06]. ARepair takes an Alloy model and a test suite especially built to alloy specification, where at least one failing test. ARepair then changes the model by applying mutations such that some failing tests become passing while passing tests stay passing. ARepair iterates until all tests pass and a repair is produced.

With regard to goal-oriented specifications, automated specification repair techniques are lacking. Goal-oriented specification considers goals as prescriptive statements of how a system should behave, while the goals are commonly written in LTL. They reflect stakeholders' understanding of what an envisioned system is intended to do. However, goals may admit subtle situations that make them diverge, e.g., when the satisfaction of some goals inhibits the satisfaction of others. The goal-conflict analysis aims at identifying, assessing, and resolving these divergences early in the development process since it enables one to improve specifications, create countermeasures, and deeply understand the roots. Despite the fact that there exist automated approaches to identify goal-conflicts, there is currently no automated approach to assist the engineer in resolving the identified divergences. This is to state that there is no automated method for repairing specifications written in LTL and in the goal-oriented specification.

1.2.2 LTL Specification Mining

The mining problem has been proposed by different approaches with different variable components [LPB15; GNR$^+$21; RGB$^+$23]. The mining problems of lightweight software representation represent a significant challenge as a research field of study. Common software representations are finite traces that represent a program or software by events captured by monitors at runtime. Furthermore, mining tools also address templates that exemplify potential structural formulas, and their instances can represent occurrences in a finite trace.

In the context of LTL specification, there are several types of LTL properties, such as safety, liveness, and fairness [OL82]. Moreover, future LTL operators generate a high level of expressiveness and complexity [SC85; BSS$^+$07]. In this manner, the LTL mining problem is a common challenge, as it allows a fast mode to specify the mentioned properties from a target system. The LTL mining problem has the potential to generate an initial specification for a target system. Moreover, it also presents the potential to complete existing LTL specifications and consequently improve them with a reduced effort.

1.2.3 Soundness of LTL Solvers

LTL solvers have been proposed to enhance or refine model checking tools [Bie21], software and hardware verification [Gup93; DKW08], and artificial intelligence reasoning [CTM$^+$17]. LTL solvers are constructed using a variety of sophisticated search algorithms and data structures.

In this manner, LTL solvers have been proposed to address the satisfiability problem. In that effort, the LTL solvers were designed around several algorithms and data structures. Among the ends of additional effort, developments also address performance improvement [SD11], diversity of temporal operators [GGM$^+$21a], and expressiveness [Lam02]. We present effective strategies for satisfiability, including the automata-theoretic approach and tableau.

There are testing approaches for SAT and SMT solvers [BLB10; BMB$^+$18], the problem of testing LTL solvers remains largely unexplored. To the best of our knowledge, there is no approach that automatically tests LTL solvers beyond some set of static benchmark formulas [SD11]. Consequently, there is a gap in methods for purposely testing LTL solvers as black-box software.

1.2.4 Reliable Portfolio of LTL Solvers

A particular satisfiability algorithm may offer the best performance in a particular problem while it has a poor performance for another. A potential solution for the LTL satisfiability problem is the algorithm portfolio. That is, the strategy involves the execution of algorithms in parallel with the objective of achieving better distribution performance. In summary, the algorithms portfolio exploits the lack of correlation in the performance of multiple algorithms to achieve better performance. Overall, it achieves better performance in the average case. However, implementations of LTL satisfiability problems have been reported as tools that contain bugs.

Once each portfolio member exploits the optimal time performance for a specific subset of a given problem. The portfolio presents a potential risk, as combining

LTL solvers in a single portfolio may also result in the combination of previous bugs present in each one. Furthermore, the literature has documented cases where SAT and SMT are utilized as solution tools that are bug-prone [BLB10; BMB^{+}18; SYW^{+}23]. The main challenge is to build a portfolio of LTL solvers that deal with potential bugs in the solver. In addition, a reliable portfolio should have a competitive average runtime performance.

1.3 Overview of the Contributions

This work addresses the four aforementioned challenges and guides the following contributions:

ACoRe: Automated Goal-Conflict Resolution (Chapter 3). Despite the fact that there exist automated approaches to identify goal-conflicts, there is currently no automated approach to assist the engineer in resolving the identified divergences. We present ACoRe, an automated approach to resolve goal-conflicts in requirements specifications expressed in LTL. ACoRe explores syntactic modifications of the specifications to create variants that remove all the previously identified conflicts while preserving consistency. In that process, ACoRe also maintains syntactic and semantic similarity. Intuitively, syntactic similarity refers to the distance between the text representations of the original specification and the candidate repair. Semantic similarity refers to the system behavior shared by the original specification and the candidate repair, and it is computed by using model counting approximation. We evaluated ACoRe based on 25 requirements and specifications collected from the literature and benchmarks. In summary, the empirical results present a reasonable number of repairs in the Pareto-optimal set (between 1 and 8). Also, we show that ACoRe obtains better results when NSGA-III is used, as opposed to other search algorithms, measured in terms of quality indicators, while at the same time, it produces more repairs that do not introduce new conflicts.

Evaluating Specification Mining in Linear Temporal Logic (Chapter 4) Linear temporal logic (LTL) has a general set of operators that impose high expressiveness and complexity. On the basis of the high level of expressiveness, tools have been created to mine LTL properties based on trace artifacts. A trace describes the relevant events that occur or the behavior changes of concurrent systems. The LTL mining tools analyze patterns or constraints in these traces and propose LTL properties that are valid in the provided traces. However, there is a gap in the evaluation of the effectiveness and capability of mining tools. For instance, if the tools are able to propose correct properties or reach a soundness set of LTL properties. Therefore, we propose an evaluation of LTL mining tools. First, we assess them based on the capability to find a ground truth set of LTL specifications. Second, we observe the capability to cover the ground truth set of LTL specifications. Our evaluation shows that the mining tools do not find

LTL properties in a ground truth composed of formal specifications. Moreover, the experiment also points out that the mining tools have low precision in the semantic coverage of the ground truth. The contribution suggests the LTL mining specification as an incipient research field. The main challenge of automatically generating or completing the existing specification is still an unfilled challenge.

Fuzzing Linear Temporal Logic Solvers (Chapter 5). Our fuzzing technique, namely SpecBCFuzz, is based on search-based fuzzing that generates and evolves failure-inducing input by evaluating fitness functions. Among the fitness functions, the boundary conditions are extensively evaluated during the search process, which progressively resolves the boundary conditions in the LTL specifications that were provided as seeds. Essentially, a boundary condition represents a conflict condition whose occurrence results in the loss of satisfaction of the goal's properties. Thus, the evolved specifications are valid repairs for the previously identified divergences. We claim that the search process for resolving boundary conditions efficiently produces failure-inducing input. Moreover, additional fitness, such as semantic and syntactic similarities, can constrain the search space and maintain the seeds given by the specifications as representative of realistic LTL formulas. Finally, the algorithm computes the satisfiability of the modified specification. Our empirical evaluation relies on differential fuzzing testing and inconsistency patterns. The result suggests that our search-based fuzzing based on boundary condition resolution is significantly more efficient than a baseline approach (e.g., LTL probabilistic grammar fuzzing). Also, the empirical study points out a larger size effect of the inconsistency patterns identified during the empirical study execution. Furthermore, our empirical evaluation presents 16 bugs in the evaluated LTL solvers. In addition, we present performance warning patterns. They are useful for performance testing purposes. The set of warning patterns has the potential to indicate improvements in a particular solver for releases, heuristics set, or theoretical guidelines for a particular type of LTL satisfiability algorithm.

Fuzzing Linear Temporal Logic Portfolio (Chapter 6). We modify portfolio decisions to incorporate a confirmation answer. In other words, the portfolio does not provide the fastest answer. The portfolio replicates the answer when at least two solvers agree on a given input. That is, the new portfolio implements a lockstep fault-tolerant mechanism. It was formulated based on a previous study that identified bugs in LTL solvers and demonstrated that a considerable number of these bugs are not inherent to a single input across different solvers. Our empirical results present bugs in the pure portfolio, whereas a search-based fuzzing approach does not find a single bug for the reliable portfolio. Furthermore, we also evaluated the reliable portfolio in terms of performance. The empirical results present the lowest runtime for the reliable portfolio compared to a single robust LTL solver (NuSMV). In summary, a reliable portfolio is capable of increasing soundness in

LTL solvers and is also associated with a gain in performance compared to an individual LTL solver that contains a high level of soundness.

1.4 Organization of the Dissertation

Chapter 2 presents the related work of this dissertation. It covers topics of LTL and their associated challenges, such as automated repair of specification, specification mining, soundness of solvers, and a reliable portfolio of solvers.

Chapter 3 introduces ACoRe, an automated approach to resolve goal-conflicts in requirements specifications expressed in LTL. In summary, an automated specification repair approach for LTL specifications. ACoRe is a search-based approach that applies syntactic modifications of specifications to remove previously identified conflicts. ACoRe is guided by the syntactic and semantic similarity of the repair candidates and the original specifications. In addition, the search also considers the satisfiability and consistency of the repairs produced. Our results show that ACoRe is capable of producing a reasonable number of repairs.

Chapter 4 presents a new comparison of specification mining tools. In this manner, we inquire about the feasibility of mining LTL specifications based on software artifacts such as finite traces and templates for LTL properties. Consequently, the specification mining tool can also support existing specifications and improve them with new properties. Our empirical results suggest that LTL mining tools are still incipient.

Chapter 5 proposes a search-based fuzzing, named SpecBCFuzz, based on previous fitness used in automated specification repair to produce complexity inputs and evolves failure-inducing inputs. Our results show that the search process for resolving boundary conditions efficiently produces failure-inducing input. Moreover, semantic and syntactic similarities also maintain the seeds given by the specifications as representative of realistic LTL specifications.

Chapter 6 introduces modifications to the decision-making process in LTL portfolios, incorporating a confirmation answer. Therefore, the portfolio does not offer the most expedient response. The portfolio replicates the answer in instances where a minimum of two solvers concur on a given input. The formulation was based on our previous study that identified bugs in LTL solvers and demonstrated that a considerable number of these bugs are not inherent to a single input across different solvers. In summary, a reliable portfolio with a confirmation answer is capable of increasing soundness in LTL solvers and is also associated with an improvement in performance compared to an individual LTL solver that contains a high level of soundness.

Chapter 7 offers a comprehensive summary of this dissertation and suggests avenues for future research.

2 Related Work

This chapter discusses the related work to the challenges and contributions of this dissertation. Special attention was paid to covering all the published work in this dissertation. We present an overview of the related work about Linear Temporal Logic (LTL) and the research challenges associated with specification repair, specification mining, soundness of solvers, and a reliable portfolio of solvers.

Contents

2.1 Automated LTL Specification Repair 12

2.2 LTL Specification Mining 14

2.3 Soundness of LTL Solvers 16

2.4 Reliable Portfolio of LTL Solvers 17

2.1 Automated LTL Specification Repair

Formal requirements are widely studied [LFD$^+$19; vLDL98a] and used in different software domains for specifying critical software systems [AIL$^+$07; BBC$^+$00; MP95] and reactive systems [BH08; Koy92]. Several methods have been proposed to deal with these kinds of requirements (e.g. SCR [HAB$^+$05], Alloy [Jac12], Event-B [Abr10]). Much research over the last decades has demonstrated the significant advantages that goal-oriented methodologies, such as KAOS [DvLF93; vLam09] and I* [Yu97], bring to the generation of correct software requirements specification. Particularly, KAOS uses Linear Temporal Logic (LTL) for formally expressing high-level goals that are subject to several analysis, e.g., to support the derivation of software operations [AKR$^+$09; DAA$^+$14]. However, for such tasks to be successfully achieved, the goals themselves must be correct, which is not often the case and inconsistencies have to be identified and resolved.

Thus, several manual approaches have been proposed to identify inconsistencies between goals and resolve them once the requirements were specified. Among them, Murukannaiah *et al.* [MKT$^+$15] compares a genuine analysis of competing hypotheses against modified procedures that include requirements engineer thought process. The empirical evaluation shows that the modified version presents higher completeness and coverage. Despite the increase in quality, the approach is limited to manual applicability performed by requirements engineers as well previous approaches [vLDL98a].

Various informal and semi-formal approaches [HHT02; Kam09; KHG11], as well as more formal approaches [ESH14; EBM$^+$12; HKP05; HN98; SF97; NVL$^+$13], have been proposed for detecting logically inconsistent requirements, a strong kind of conflicts. In this work we focus on weak form of conflict, called divergences, that despite the goals are consistent, there exists subtle situations that lead to the identified inconsistencies.

Peng *et al.* [Kam09] introduces an automated tool to support analysis of requirements and traceability between different representations: textual, visual, informal and formal. However, it does not focus on identifying weak inconsistencies like goal-conflicts boundary conditions or help in resolving the conflicts are limited to conflict constraints, separated execution of goals, and different requirements in the convergence domain. Furthermore, the approach is specific since it deals with the convergence of goal models. That is, it is not applied in a general-purpose as our proposed approach.

Recent approaches have been introduced to automatically identify goal-conflicts. Degiovanni *et al.* [DRA$^+$16] introduced an automated approach where boundary conditions are automatically computed using a tableaux-based LTL satisfiability checking procedure. This approach exploits a complex logical algorithm to generate the tableau for the specification (an abstract semantic tree that aims to capture all the models satisfying the formula) to generate the boundary conditions, but it exhibits serious scaliability issues, limiting its applicability. To deal with this issue, Degiovanni *et al.* [DMR$^+$18] proposes a genetic algorithm that changes the LTL formula syntax in order to find boundary conditions by a genetic algorithm. The empirical evaluation showed that it can successfully compute several boundary conditions, requiring much less computational time, outperforming the tableaux-based approach.

Other approaches were proposed to prioritize and assess the identified boundary conditions. For instance, Luo *et al.* [LWS$^+$21] introduces the notion of contrasty metric in order to select (a few) non-redundant boundary conditions characterizing, in some sense, different kind of conflicts. The work of Degiovanni *et al.* [DCA$^+$18] proposes to use LTL model counting to estimate the likelihood and severity of boundary conditions that help the engineer to prioritize some conflicts among others.

Regarding specification repair approaches, Wang *et al.* [WSK19] recently introduced ARepair, an automated tool repair a faulty model formally specified in Alloy [Jac06]. ARepair takes an Alloy model and a test suite, possibly generated with AUnit [SWZ$^+$17], where at least one failing test. ARepair then changes the model, by applying mutations, such that some failing tests become passing while passing tests stay passing. ARepair iterates until all tests pass and a repair is produced. In the case of ACoRe, instead of failing tests, the identified goal conflicts are the ones that guide the search, and candidate repairs are aimed to be syntactic and semantically similar to the original buggy specification.

More related to our approach, Alrajeh *et al.* [ACvL20] recently introduced an automated approach to refine a goal model when the environmental context changes. That is, if the domain properties, then this approach will propose changes in the goals to make them consistent with the new domain. The adapted goal model is generated using a new counterexample-guided learning procedure that ensures the correctness of the updated goal model, preferring more local adaptations and more similar goal models. In our work, the domain properties are not changed and the adaptions are made in order to remove the identified inconsistencies, and instead of counter-examples.

Several approaches have been presented for automatically synthesizing a correct-by-construction implementation for a reactive system from an LTL specification [EC82; MW84; PR89]. Frequently, these specifications present some imperfections that make them *unrealisable*, i.e., no implementation can be synthesized that satisfies them. Majority of the approaches that aim to deal with this issue, focused on learning missing assumptions on the environment that make them unrealisable [AMT13; CA17; CHJ08; MRS19]. More recently, Brizzio *et al* [BCP$^+$23] published a report in which an automated approach for repairing unrealizable

LTL specifications is proposed. The approach aims to transform unrealizable specifications into realizable ones by performing syntactic modifications to the LTL formulations. Similarly to the report of Brizzio *et al.* [BCP+23], ACoRe adopted the novel model counting approach to compute the semantic similarity between the specifications, and adapted the evolutionary operators to perform variants in the LTL specifications. However, the notion of repair for Brizzio *et al.* [BCP+23] requires to obtain a realizable specification, which is very general and not necessarily leads to quality synthesized controllers [DBP+13; MR16]. In this work, the definition of repairs is more fine grained and focused on removing the identified conflicts, which potentially lead to interesting repairs as we showed in our empirical evaluation. Moreover, while Brizzio *et al.* [BCP+23] implemented a search similar to our weighted-based genetic algorithm (WBGA), ACoRe implements three different novel multi-objective optimizations algorithms, including AMOSA and NSGA-III. Interestingly, these consider the dominance notion between candidates and guide the search throughout the Pareto-optimal set, leading to good repairs (particularly, NSGA-III was the best performing in our experiments). The Pareto-optimal set is a powerful way to reduce the number of repairs to present to the engineer requiring analysis.

2.2 LTL Specification Mining

Several works present solutions to the LTL mining solver and correlated topics.

Kang *et al.* [KL21] introduce an approach named DICE (Diversity through Counter-Examples). Dice explores adversarial specification mining, since it guides automated test generation for counterexamples test cases for intermediary specifications. To summarize, DICE is divided into two parts. They are DICE-Tester and DICE-Mining. Before calling DICE-Tester, the pre-processing generates automated test for a program under analysis. The expected results are a finite trace that represents the program under analysis. Based on six pre-established templates of temporal properties, a temporal specification is automatically produced. Following the generation of LTL specifications from the execution of a test suite, DICE-Tester applies an adversarial approach to guide the generation of tests, searching to identify counterexamples that would invalidate some temporal properties. Ideally, the counterexamples represent lacks in the diversity of the initial test suite.

In addition to that, DICE-Miner infers finite state automaton using the new traces produced by the refined tests in DICE-Tester, Moreover, DICE-Miner also guides the inferring process by the temporal specifications. Therefore, DICE receives a program under analysis and an initial test suite as input, generating a finite state automaton as output. Regarding the limits of the LTL mining problem, DICE is coupled to six pre-established templates of temporal properties. Moreover, DICE uses the mined temporal properties as an intermediary representation that is transformed into a test goal for the search process.

Raha *et al.* [RRF^{+}22] present the Scarlet tool. It considers the LTL mining problem in fragments of LTL, while it classify traces. Ideally, a separation formula must satisfy the positive traces and not the negative traces. Scarlet is divided into two parts. They are searching for directed formulas and a subsequent part of combining directed formulas. Searching for directed formulas applies a dynamic programming algorithm that iteratively generates formulas and evaluates them according to finite traces provided by users. In the first, the algorithm defines an order with two parameters: length and width. The length is the number of conjunctions of atomic propositions in a generated formula, while the width is the maximum of the atomic propositions of the mentioned conjunctions. The algorithm interatively increases the parameters length and width in the generated formulas. As a dynamic programming algorithm, it stores a partial evaluation of the association mentioned of the LTL formula generated with the trace in a dynamic table.

The result of searching for directed formulas is smaller formulas with higher association with finite traces. Besides that, the main LTL mining algorithm executes the second part named combining directed formulas. It runs an off-the-self decision tree algorithm, and a greedy algorithm for the boolean subset cover problem. In summary, given the small formulas generated in the first part, the combining directed formulas searches for a boolean combination of some of the formulas that satisfy all positive and non-negative traces. To summarize, Scarlet implements a pattern match approach for the LTL mining problem. Regarding the additional features, Scarlet works with positive and negative finite traces, as mentioned. Moreover, noisy data are also accepted by Scarlet that the main algorithm builds an approximate classifier. With regard to Scarlet limits, it considers the mining LTL formulas without the U-operator. Thus, it forms a very constrained LTL grammar.

Roy *et al.* [RGB^{+}23] present the prototype named samp2symb. It mines deterministic finite automata and LTL properties. We concentrate on the main features and differences in the case of LTL property mining. The prototype samp2symb presents a mining algorithm based on semi-symbolic and counterexample-guided approaches. The semi-symbolic algorithm is of the type constraint solver since converts part of the search process into logical constraints. Interactively, the semi-symbolic algorithm instantiates a search algorithm that relies on a set of negative cases. It uses a current LTL formula and the negative case to build a new LTL formula.

The new concrete LTL formula is generated and should satisfy the current LTL formula and does not hold for the sample words of the negative cases. For the generation of words, the semi-symbolic algorithm builds a finite automata from the LTL formula and then conducts a breadth-first search. The construction of the new concrete LTL formula is a conjunction of logical constraints. The structural and semantic constraints as mentioned in Section 4.4, the negative and positive words constraints, and the words that distinguish the new contract LTL formula and the current LTL formula.

The counterexample-guided algorithm stores a set of negative words and a set of discarded LTL formulas. Interactively, the algorithm builds a new concrete LTL formula, and it does not hold the words in the negative set. Moreover, it is not one of the discarded LTL formulas. Finally, the new concrete LTL formula is built based on a model of the current LTL formula. Essentially, the algorithm stores the cases in which the new concrete LTL formula does not hold on a subset of the set of words generated by current LTL formula. The current LTL formulas are added to the discarded set. In this scenario, the counterexample-guided algorithm is associated with a pattern match solution as mentioned in Section 4.3.

Texada and Sample2LTL were discussed in Section 4.3 and Section 4.4. They are complete LTL mining tools insofar as they deal with finite traces and LTL templates without additional constraints. Moreover, they are implemented in available and reliable tools.

2.3 Soundness of LTL Solvers

Fuzz testing is a traditional technique and is successful for solvers testing. Among them, are boolean satisfiability (SAT), quantified boolean satisfiability (QBF), and satisfiability modulo theories (SMT).

Brummayer [BLB10] presented successful fuzzes for SAT and QBF solvers. The fuzzes are based on black-box grammar-based fuzzing. Among them, CNFuzz generates CNF formulas based on a traditional CNF grammar, depending on parameters such as maximum layers, width, and variables. The specific number per each formula is chosen randomly in a pre-defined range. Moreover, FuzzSAT represents the formula in a random boolean circuit (RBC), which is a directed acyclic graph. FuzzSAT builds the RBC by randomly selecting a boolean operator from a common set of operators. The operands include variables, and the negation happens with a probability of 1/2. The stop condition depends on the minimum reference time for each variable. The additional parameter includes the number of input nodes in the graph. QBFuzz works similarly. Additional parameters include the ratio of existential variables in the formulas. The empirical evaluation shows the capability to find bugs in SAT and QBF solvers.

SMT solvers have a large variety of software testing approaches to test them. StringFuzz randomly generates and transforms string constraints [BMB^{+}18]. To generate, the fuzzing approach integrates generators such as random string, random regular expression, combines of random extract, overlap expressions, and equality of combines. After the generation step, transformers can be applied for different reasons. For instance, the transformers can increase the complexity of generated inputs or preserve satisfiability when a subset of transformers are applied in the fuzzing testing. The general set of transformers contains replacement literals and operators, swaps non-leaf nodes with leaf nodes, permutes the alphabet, reverses all string literals, and combines arguments. Moreover, StringFuzz can also include a seed-based strategy to improve fuzzing testing. The seed-based strategy relies on including realistic regular expression sets instead of random ones. The SMT language is based on SMT-LIB [BST10]. Bugariu and Muller's approach generates more complex string operations [BM20]. Moreover, the proposed approach can generate inputs that are satisfiable or unsatisfiable by construction. The formulas are used as test oracles. Other fuzzing strategies are also commonly found. For instance, STORM adopts seed-based and mutation fuzzing techniques [MCW+20].

Moreover, several other fuzzing approaches have been proposed for testing SMT solvers. For example, HistFuzz explores the usage of the seed-based strategy by skeletons created from the bug history [SYW+23]. Moreover, the approach also derives association rules from historical formulas. Thus, skeletons and association rules are used to guide the fuzzing test. Furthermore, DIVER combines traditional fuzzing strategies (e.g., mutation), oracle-guided approaches (seed-based and its abstract values for SAT preservation), and differential testing [KSO23]. In general, different fuzzing tests can find bugs in popular SMT solvers such as Z3 [dMB08], CVC5 [BBB+22] and dReal [GKC13].

Regarding LTL solvers, Schuppan and Darmawan [SD11] conducted an empirical evaluation based on benchmark families. Specifically, the benchmark formulas were extracted from the four previous empirical studies that evaluated the performance of new LTL solvers or compared their results. In addition, the authors added new formulas in the benchmark to improve the challenge and also scaled up some formulas. For example, the authors included new formats of LTL formulas and liveness conditions to cover non-trivial behavior. From a different perspective of our evaluation, they compared the performance of LTL solvers, while our work evaluated the soundness of each solver by differential fuzzing approaches. Besides that, the benchmark families have found inconsistencies. The authors reported a few pairs of them for some LTL solvers. However, the authors do not discuss the characteristics related to bugs or even features in the benchmark families that handle error prune inputs.

2.4 Reliable Portfolio of LTL Solvers

Many portfolio approaches have been proposed for satisfiability problem. Among them, propositional satisfiability and their parallel competition [FHI+21]. LTL portfolio was a constrained environment. We may mention two previous work that consider LTL portfolio based on the diversity principle: (i) Shuppan's benchmark and (ii) Polsat.

Schuppan's benchmark assesses LTL solvers based in a new families of benchmarks. The conclusion of a performance evaluation based on the families of benchmarks indicates that there is not a single LTL solver that answers fastest in a widely point of view [SD11]. The result points out that LTL solvers should be combined based on performance analysis. The combination based on performance analysis reaches diversity of algorithms for LTL safisfiability. Despite of the combination of LTL solvers reach a better performance in terms of score competition, the assess is absent of corrects evaluation and the possibility to combine previous bugs per each solver.

The same happens for Polsat [LPZ+13] that implements a pure portfolio with four solvers: NuSMV (BMC and BDD versions), Aalta, pltl, and TRP++. The portfolio was built in terms of the diversity principle that considers LTL satisfiability strategies such as multi pass tableau algorithm and reduction to SAT-solvers. The score competition suggest Polsat as more efficient than single LTL solvers. However, the soundness evaluation of the Polsat is not take into account.

In the case of testing solvers, fuzzing approaches have been proposed. Among them, Carvalho *et al.* [CDC+24] presents SpecBCFuzz for fuzzing LTL solvers. It is a search-based fuzzing that considers formal specifications written in LTL as seeds. In the search process, SpecBCFuzz conducts an evolution search with mutation and crossover and optimizes fitness related to semantic and syntax distance between transformed and original specification. Moreover, the SpecBCFuzz also optimizes the satisfiability of the transformed specification and boundary conditions (representing conflicts and divergences between properties) as a metric to maintain the search in the boundaries of SAT and UNSAT.

For SAT solvers, Brummayer *et al.* [BLB10] presents successful fuzzes for SAT and QBF solvers. For example, FuzzSAT represents the formula in a Random Boolean Circuit (RBC). FuzzSAT builds the RBC by randomly selecting a boolean operator from a common set of operators. The operands are variables and their negations. The stop condition depends on the minimum reference time of each variable. The additional parameter contains the number of input nodes of the directed acyclic graph representing the RBC. QBFuzz works similarly. The additional parameter is the ratio of the existential variables in the formulas. As in the previous case, the fuzzing approach considers correctness, but it does not consider the performance of SAT solvers.

SMT solvers have a large variety of software testing approaches to test them. StringFuzz randomly generates and transforms string constraints [BMB+18]. To generate, the fuzzing approach integrates generators such as random string, random regular expression, combines of random extract, overlap expressions, and equality of combines. Other fuzzing strategies are also commonly found. For instance, STORM adopts seed-based and mutation fuzzing technique [MCW+20].

3

Automated Goal-Conflict Resolution

Goals are prescriptive statements of how a system should behave. They reflect the understanding of stakeholders of what an envisioned system is intended to do. However, goals may admit subtle situations that make them diverge, for example, when the satisfaction of some goals inhibits the satisfaction of others. The goal-conflict analysis aims at identifying, assessing, and resolving these divergences early in the development process, as it enables a greater understanding of the specifications, creating countermeasures, and understanding the root causes of problems. Despite the fact that there exist automated approaches to identify goal-conflicts, there is currently no automated approach to assist the engineer in resolving the identified divergences. We present ACoRe, an automated approach to resolve goal-conflicts in requirements specifications expressed in Linear Temporal Logic (LTL). ACoRe explores syntactic modifications of the specifications to remove all the previously identified conflicts, while preserving consistency. Semantic similarity refers to the system behavior shared by the original specification and the candidate repair, and it is computed by using model counting approximation. We evaluated ACoRe on 25 requirements specifications collected from the literature and benchmarks. In summary, the empirical results present a reasonable number of repairs in the Pareto-optimal set (between 1 and 8). Also, we show that ACoRe obtains better results when genetic algorithms (e.g., NSGA-III) are used, as opposed to other search algorithms, measured in terms of quality indicators, while at the same time, it produces more repairs that do not introduce new conflicts.

Contents

3.1 Introduction 23
3.2 Background 25
3.3 Goal-oriented Specifications and Conflicts 29
3.4 ACoRe: Automated Goal-Conflict Resolution 35
3.5 Research Questions 41

3.6 Experimental Setup 41
3.7 Experimental Results 45
3.8 Threat to Validity 51
3.9 Conclusion 52

3.1 Introduction

The requirements analysis phase focuses precisely on the elicitation, understanding, and specification of requirements documents [IEE98] that describe in detail the desired system behavior. Among requirements engineering methods, the goal-oriented requirements engineering (GORE) methodologies [DvLF93; vLam09], provide an intuitive way of modeling and analyzing the objectives of the envisioned system. In these approaches, requirements are organized around the notion of *goals*, prescriptive statements that specify what the system to be developed should do.

In GORE methodologies, goals are the subject of several activities, including goal decomposition and refinement [DvLF93; vLam09; vLDL98a], assigning to agents for their realization [AKR+09; DAA+14; Let02]. They are also assessed in many ways, which include assessments that evaluate for feasibility [NSG+18], and assessments for evaluating potential threats or risks [CvL15; DCA+18]. In particular, risk analysis deals with this last issue in various ways, including goal-conflict analysis [vLL00].

The *conflict* represents a condition that, when present, makes the goals inconsistent. The goal conflict analysis consists of three main stages: (1) the *identification* stage, which involves the identification of conflicts between goals; (2) the *assessment* stage, which aims to assess and prioritize the identified conflicts according to their likelihood and severity; and (3) the *resolution stage*, where conflicts are resolved by providing appropriate countermeasures and, consequently, transforming the goal model, guided by the criticality level obtained during the evaluation.

Techniques have been proposed to assist engineers in identifying and assessing goal-conflicts. Among them, Van Lamsweerde *et al.* [vLDL98a] proposed a set of predefined syntactic patterns to identify goal-conflicts that only apply to some specific goal expressions. Recently, Degiovanni *et al.* [DCA+18; DRA+16] presented two automated approaches for goal-conflict identification, which combine novel logical manipulations of specifications and modern search-based engines. In addition, two recent approaches deal with the problem of assessing goal-conflicts, namely, the work of Degiovanni *et al.* [DCA+18] proposed to use model counting for estimating goal-conflicts likelihood and severity, while the work of Luo *et al.* [LWS+21] proposed a goal-conflict selection technique by studying their overlapping from a logical point of view.

Thus, we proposed an automated approach that deals with the goal-conflict resolution stage [CDB+23], that is, ACoRe. In general, ACoRe receives as input $Dom = \{Dom_1, \ldots, Dom_n\}$, $G = \{G_1, \ldots, G_m\}$, and $BC = \{BC_1, \ldots, BC_k\}$, where the domain properties (Dom) are descriptive statements that capture the domain of the problem world. For example, it can describe physical properties. In the other sense, goals (G) are prescriptive statements that the system should achieve or maintain within a given domain. For example, what the stakeholders whish. Finally, the boundary conditions (BC) set represents divergences between the domain and goals properties.

The initial set of domain properties and the elicited goals are formally expressed in Linear Temporal Logic (LTL) [MP92], and the set of identified goal-conflicts, also expressed in LTL, respectively. ACoRe then generates a set of *repairs* $R = \{R_1, \ldots, R_z\}$ such that every specification R_i is consistent and removes all the previously identified conflicts BC. ACoRe employs modern search-based algorithms to efficiently explore syntactic variants of goals (throughout the application of genetic operators) to generate repairs that are in some sense similar to the original specification. In summary, ACoRe considers both syntactic and semantic similarity between G and the candidate repairs and implements a multi-objective fitness function to guide the search. The syntactic similarity is computed as the Levenshtein edit distance [LY19], while semantic similarity is computed by a novel LTL model counting approximation approach we recently developed in [BCP$^+$23].

Regarding the experimental evaluation, we study the effectiveness of ACoRe in producing repairs that remove all identified goal-conflicts (RQ1) and measure its similarity concerning the ground truth, i.e., to the manually written repairs (RQ2). Our results present that ACoRe generates more non-dominated repairs when adopting genetic algorithms (NSGA-III or WBGA) than AMOSA or unguided search. The genetic algorithms outperform the others in 21 out of 25 cases.

Overall, the algorithms of ACoRe produce an acceptable number of repairs in the Pareto-optimal set (between 1 and 8) that can easily be explored by the engineer to select and validate the most appropriate one. The evaluation to answer RQ2 points out that the genetic algorithms can resemble more ground truth repairs than AMOSA or unguided search. Moreover, in the cases in which ACoRe fails to resemble the ground truth, we observe that the repairs generated by the genetic algorithms are closer to the manually written repairs, measured in terms of the quality indicators hypervolume and inverted generational distance, outperforming AMOSA and unguided search.

We also performed an objective comparison. First, we study if the repairs are likely to introduce new goal-conflicts (RQ3), which would make them unsuitable repairs. Our results show that ACoRe adopting WBGA and NSGA-III obtains a statistically significant difference compared to the other algorithms when we consider the number of repairs that do not introduce new boundary conditions. Second, we compare the repairs produced by each algorithm by using standard quality indicators (RQ4), such as the hypervolume (HV) and inverted generational distance (IGD). NSGA-III outperforms the other algorithms since it produces Pareto-optimal sets with significantly higher quality indicators (HV and IGD).

During our experimental evaluation, we *extended* the previous experiment in many different ways. Among them, RQ2 includes the distance from the proposed repairs and the set of ground truth. RQ3 is entirely new and presents the analysis of introduced goal-conflicts, while RQ4 extends the analysis and discussion about the Pareto front analysis based on quality indicators (HV and IGD).

The work is organized as follows. Section 3.2 presents the background about LTL and multi-objective optimization. Section 3.3 focuses on GORE and illustrative examples that show common tasks in the goal-analysis and goal-resolution stages. Section 3.4 presents ACoRe in detail. Section 3.5 presents the research questions, while Section 3.6 discusses the empirical design decisions. Section 3.7 presents the empirical results. Section 3.8 discusses threats to the validity of our work. Finally, we discuss related work in Section 2.1 and conclude in Section 3.9.

3.2 Background

3.2.1 Linear-Time Temporal Logic

Linear Temporal Logic (LTL) is a logical formalism widely used to specify reactive systems [MP92]. Several GORE methodologies (e.g., KAOS) have also adopted LTL to formally express requirements [vLam09] and take advantage of the powerful automatic analysis techniques associated with LTL to improve the quality of their specifications.

LTL formulas are inductively defined using the standard logical connectives and the temporal operators to describe future events. For example, the Boolean connective ($\vee$) and the traditional definitions for *true* and *false*. For future events, we commonly define next ($\bigcirc$) and until ($\mathcal{U}$). For a formula φ and position $i \geq 0$, we say that φ holds at position i of σ. Thus, we write $\sigma, i \models \varphi$. Thus, LTL formulas are defined as follows:

Definition 1 (LTL Syntax). *Let AP be a set of propositional variables:*

(a) constants true and false are LTL formulas;

(b) every $p \in AP$ is a LTL formula;

(c) $p \in AP$ and $\sigma, i \models p$ iff $p \in \sigma_i$;

(d) $\sigma, i \models \neg\varphi$ iff $\sigma, i \not\models \varphi$;

(e) $\sigma, i \models \varphi \vee \phi$ iff $\sigma, i \not\models \varphi$ or $\sigma, i \models \phi$;

(f) $\sigma, i \models \bigcirc\varphi$ iff $\sigma, i+1 \models \varphi$;

(g) $\sigma, i \models \varphi \mathcal{U} \phi$ iff there exists $n > i$ such that $\sigma, n \models \phi$ and $\sigma, m \models \varphi$ for all m, $i \leq m < n$;

(h) if φ and ψ are LTL formulas, then are also LTL formulas $\neg\varphi$, $\varphi \vee \psi$, $\bigcirc\varphi$, and $\varphi\mathcal{U}\psi$.

We also consider other typical connectives and operators, such as, $\wedge$, $\Box$ (always), $\Diamond$ (eventually) and $\mathcal{W}$ (weak-until), that are defined in terms of the basic ones. That is, $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$, $\Diamond\phi \equiv \mathit{true}\,\mathcal{U}\,\phi$, $\Box\phi \equiv \neg\Diamond\neg\phi$, and $\phi\mathcal{W}\psi \equiv (\Box\phi) \vee (\phi\mathcal{U}\psi)$.

3.2.2 Model Counting

The *model counting* problem consists of calculating the number of models that satisfy a formula. Since the models of LTL formulas are infinite traces, it is often the case that analysis is restricted to a class of canonical *finite* representations of *infinite* traces, such as lasso traces or tree models. In particular, this is the case for bounded model checking, for instance [BCC+99b].

Definition 2 (Lasso Trace). *A lasso trace σ is of the form $\sigma = s_0 \dots\ s_i\ (s_{i+1} \dots s_k)^\omega$, where the states $s_0 \dots s_k$ correspond to the* base *of the trace, and the loop from state s_k to state s_{i+1} is the part of the trace that is repeated infinitely many times.*

For example, an LTL formula $\Box(p \vee q)$ is satisfiable, and another satisfying lasso trace is $\sigma_1 = \{p\}; \{p, q\}^\omega$, where in first state p holds, and from the second state both p and p are valid forever. Notice that the base in the lasso trace σ_1 is the sequence containing only state $\{p\}$, while the state $\{p, q\}$ is the sequence in the loop part.

Definition 3 (LTL Model Counting). *Given an LTL formula φ and a bound k, the (bounded)* model counting *problem consists in computing how many lasso traces of at most k states exist for φ. We denote this as $\#(\varphi, k)$.*

Since existing approaches for computing the exact number of lasso traces are ineffective [FT14], Brizzio *et al.* [BCP+23] recently developed a novel model counting approach that approximates the number (of prefixes) of lasso traces satisfying an LTL formula. Intuitively, instead of counting the number of lasso traces of length k, the approach of Brizzio *et al.* [BCP+23] aims at approximating the number of bases of length k corresponding to some satisfying lasso trace.

Definition 4 (Approximate LTL Model Counting). *Given an LTL formula φ and a bound k, the approach of Brizzio et al. [BCP+23] approximates the number of bases $w = s_0 \dots s_k$, such that for some i, the lasso trace $\sigma = s_0 \dots\ (s_i \dots s_k)^\omega$ satisfies φ (notice that the prefix w is the base of σ). We denote as $\#\textsc{Approx}(\varphi, k)$ to the number returned by this approximation.*

ACoRe uses the #Approx model counting to compute the semantic similarity between the original specification and the candidate repairs.

3.2.3 Multi-Objective Optimization Algorithms

Several software engineering problems have been addressed by optimization algorithms [HJ01], where such techniques could provide solutions to the difficult problems of balancing competing constraints. Harman and Jones [HJ01] coined the term Search-Based Software Engineering (SBSE) to describe these approaches. Among them, many software engineering problems are multi-objective [HMZ12], in other words, the problems contain more than one fitness function to be optimized.

In contrast to a single-objective optimization problem, where there is a global optimum solution, in a *multi-objective optimization (MOO) problem*, there is a set of solutions called the *Pareto-optimal* (PO) set, which are considered equally important since all of them constitute global optimum solutions. Typically, MOO algorithms evolve the candidate population with the aim to converge to a set of non-dominated solutions as close to the true PO front as possible and maintain as diverse a solution set as possible.

ACoRe implements four (4) multi-objective optimization algorithms that we proceed to describe.

NSGA-III

SpecBCFuzz integrates the Non-Dominated Sorting Genetic Algorithm III (NSGA-III) [DJ14] approach. It is a genetic algorithm, and the offspring is generated by applying mutation and crossover operators. In each iteration, the fitness values for each individual are computed and the Pareto dominance relation between them is approximated. NSGA-III uses this relation to create a kind of partition of the population in terms of the non-dominated level of the individuals. For instance, Level-1 contains non-dominated individuals and Level-2 contains the resulting non-dominated elements when all the individuals from Level-1 are not considered. Thus, NSGA-III selects individuals per non-dominated level to diversify the exploration and reduce the number of solutions in the final Pareto-optimal set since when many objectives are used, the Pareto-optimal set can be huge. In the end, NSGA-III returns the Pareto-optimal repairs that resolved all the goal conflicts given as input.

AMOSA

Archived Multi-objective Simulated Annealing (AMOSA) [BSM+08] is an adaptation of the simulated annealing algorithm [KGV83] for multi-objectives. The algorithm maintains a population size of 1, i.e., it only analyses one (current) individual per iteration, and a new individual is created by the application of the disturbance operator previously defined. In general, AMOSA has two particular features that make it promising for our purpose. During the search, it maintains an "archive" with the non-dominated candidates explored so far, which makes them more likely to be present in the final Pareto-optimal set. Moreover, when a new individual is created, two situations can happen. Whether the new individual dominates the current one, it then survives to the next iteration, the current element is discarded, and the archive is updated in case the new individual is in non-dominated (dominated elements are removed from the archive). Otherwise, whether the new individual is worse than the current one, still in can be selected among the current individual with some probability that depends on the "temperature" (function that decreases over the time). At the beginning, the temperature is high, then new individuals with worse fitness than the current element are likely to be selected, but this probability decreases over the iterations. This strategy helps to avoid local maximums and explore more diverse potential solutions. Finally, it analyzes all the individuals in the archive to check which ones are repairs, those individuals with objective *ResolvedBCs* equal 1 (i.e., that resolve all the boundary conditions).

WBGA or Traditional Genetic Algorithm

ACoRe also implement *Weight-based genetic algorithm* (WBGA) [Hol92]. Among the reasons, the WBGA allows one to control the fitness proportions as simple parameters. New individuals are generated by applying both the mutation and crossover operators introduced before.

Let S be the specification given as input. For each individual cR, WBGA computes the fitness value for each objective and combines them into a single fitness f defined as:

$$\begin{aligned} f(S, cR) =& \alpha * Status(cR) + \beta * ResolvedBCs(cR) + \\ & \gamma * Syntax(S, cR) + \delta * Semantic(S, cR) \end{aligned}$$

where weights $\alpha = 0.1$, $\beta = 0.7$, $\gamma = 0.1$, and $\delta = 0.1$ are defined by default, but they can be configured to other values if the engineer wishes.

Currently, the ACoRe implementation of WBGA uses the best-selector. This means that ACoRe sorts all the individuals in the population set according to their fitness value (descending order) and selects the top 100 individuals to survive to the next iteration (this is a parameter that can also be configured). All the solutions explored during the search are saved (a kind of archive) and reported. Later, further analysis is performed to prioritize only the repairs in the Pareto optimal set.

Unguided Search

This approach does not use any of the objectives to guide the search. In Figure 3.3, the unguided search starts by creating the initial population that consists of different individuals generated by successive application of the mutation operator described previously. In the end, it evaluates the four objectives to check which ones are actually a repair for each created candidate repair cR, that is, it resolves all the conflicts BC given as input. Repairs can later be prioritized by computing the Pareto-optimal set according to their fitness values of syntactic and semantic similarity.

3.3 Goal-oriented Specifications and Conflicts

A *conflict* essentially represents a condition whose occurrence results in the loss of satisfaction of the goals, i.e., that makes the goals *diverge* [vLL98]. Formally, it can be defined as follows.

Definition 5 (Goal Conflicts). *Let $G = \{G_1, \ldots, G_n\}$ be a set of goals and Dom be a set of domain properties. Goals in G are said to diverge if and only if there exists at least one Boundary Condition (BC), such that the following conditions hold:*

- *logical inconsistency:*

$$\{Dom, BC, \bigwedge_{1 \le i \le n} G_i\} \models \mathit{false}$$

- *minimality: for each $1 \le i \le n$ holds:*

$$\{Dom, BC, \bigwedge_{j \ne i} G_j\} \not\models \mathit{false}$$

- *non-triviality:*

$$BC \ne \neg(G_1 \wedge \ldots \wedge G_n)$$

In summary, a BC captures a particular combination of circumstances in which the goals cannot be satisfied as a whole. The first condition establishes that, when *BC* holds, the conjunction of goals $G_1, \ldots, G_n$ becomes inconsistent. The second condition states that if any of the goals are disregarded, then it cannot be *false* (it has to be consistent). The third condition prohibits a boundary condition to be simply the negation of the goals.

The initial specifying process, goal-conflict analysis [vLam09; vLDL98a] deals with conflicts through three main stages: (1) The goal-conflicts identification phase consists in generating boundary conditions that characterize divergences in the specification; (2) The assessment stage consists in assessing and prioritizing the identified conflicts according to their likelihood and severity; (3) The resolution stage consists in resolving the identified conflicts by providing appropriate countermeasures. Let us consider the following example to illustrate the goal-conflict analysis problem.

[Mine Pump Controller - MPC] Consider the Mine Pump Controller (MPC) widely used in related works that deal with formal requirements and reactive systems [KMS+83]. The MPC describes a system that is in charge of activating or deactivating a pump (p) to remove the water from the mine in the presence of possible dangerous scenarios. The MP controller monitors environmental magnitudes related to the presence of methane (m) and the high level of water (h) in the mine. Maintaining a high level of water for a while can cause flooding in the mine, while methane can cause an explosion if the pump is turned on. Hence, the specification for the MPC is as follows:

$$Dom : \Box((p \vee \bigcirc(p)) \rightarrow \bigcirc(\bigcirc(\neg h)))$$
$$G_1 : \Box(m \rightarrow \bigcirc(\neg p)) \qquad G_2 : \Box(h \rightarrow \bigcirc(p))$$

The domain property *Dom* describes the impact of switching on the pump (p) on the environment. For example, if the pump is turned on for 2 unit times, then the water will decrease, and the level will not be high ($\neg h$). Goal G_1 expresses that the pump should be off when methane is detected in the mine. Goal G_2 indicates that the pump should be on when the water level is above the high threshold.

Notice that this specification is consistent, for instance, in cases in which the level of water never exceeds the high threshold. However, approaches for goal-conflict identification, such as the one of Degiovanni *et al.* [DCA+18], can detect a conflict between goals in this specification. The identified goal-conflict describes a divergence situation in cases in which the level of water is high, and methane is present at the same time in the environment. Switching off the pump to satisfy G_1 will result in a violation of goal G_2; while switching on the pump to satisfy G_2 will violate G_1. This divergence situation evidences a conflict between goals G_1 and G_2 that is captured by a boundary condition such as $BC = \Diamond(h \wedge m)$.

Identified conflicts need to be resolved. Despite the recent surge of several automated approaches to assist engineers in identifying [DMR+18; DRA+16; vLDL98a] and assessing [DCA+18; LWS+21] goal-conflicts, currently, no tool supports the resolution stage. ACoRe aims at resolving goal-conflicts by exploring variants of the original specification. It relies on similarity metrics to produce repairs that are syntactically or semantically similar to the original specification.

3.3.1 The Goal-Conflict Resolution Problem

We define what is a repair and when a goal-conflict is resolved. The *resolution* stage aims at removing the identified goal-conflicts from the specification, for which it is necessary to modify the current specification formulation. This may require weakening or strengthening the existing goals or even removing some and adding new ones.

Definition 6 (Goal-Conflict Resolution)**.** *Let $G = \{G_1, \ldots, G_n\}$, Dom, and BC be a set of goals, the domain properties, and an identified boundary condition,* *respectively. We say that a repair $R = \{G'_1, \ldots, G'_m\}$ resolves goal-conflict BC, if and only if, the following condition holds:*

$$\{Dom, BC, \bigwedge_{1 \leq i \leq m} G'_i\} \not\models \mathit{false} \qquad (\mathit{logical\ consistency})$$

Intuitively, the refined goals in R know exactly how to deal with the divergence situation BC in the domain Dom. BC does not prohibit any more the satisfaction of the goals in R. This definition can be straightforwardly extended to a set of identified-goal conflicts (R resolves a set $\{BC_1, \ldots, BC_k\}$ of boundary conditions if it resolves every BC_i, for $1 \leq i \leq k$).

To generate a repair R from the initial specification, ACoRe will explore the syntactic modifications of the goals from G, leading to newly refined goals in R, until it removes the logical inconsistency with conflict BC. Thus, $R \wedge BC \wedge Dom$ becomes consistent.

It is natural to expect that the refined goals in R maintain some form of similarity to the original ones in G. ACoRe will focus on the syntactic and semantic similarities of candidate repairs w.r.t. the original specification. ACoRe integrates the similarity metrics, previously defined in Section **??**, in multi-objective optimization search algorithms that will guide the search to produce repairs that are syntactically and semantically similar to the original specification.

3.3.2 Motivating Examples

Recall that in Example 3.3, the boundary condition $BC = \Diamond(m \wedge h)$ was identified for the MPC, which coincides with the one manually identified in [Let01]. To resolve this conflict, Letier [Let01] proposes to refine the goal G_2, by weakening it, requiring to switch on the pump when the level of water is high and no methane is present in the environment.

[Ground-truth 1 - MPC]

$$\begin{aligned} Dom :& \Box((p \vee \bigcirc(p)) \rightarrow \bigcirc(\bigcirc(\neg h))) \\ G'_1 :& \Box(m \rightarrow \bigcirc(\neg p)) \\ G_2 :& \Box(h \wedge \neg m \rightarrow \bigcirc(p)) \end{aligned}$$

With a similar analysis, another repair is obtained by weakening G_1, which requires the pump to be turned off when methane is present and the water level is not high.

[Ground-truth 2 - MPC]

$$\begin{aligned} Dom :& \Box((p \vee \bigcirc(p)) \rightarrow \bigcirc(\bigcirc(\neg h))) \\ G'_1 :& \Box(m \wedge \neg h \rightarrow \bigcirc(\neg p)) \\ G_2 :& \Box(h \rightarrow \bigcirc(p)) \end{aligned}$$

Figure 3.1 shows in blue the syntactic and semantic similarities between the repairs manually reported in [Let01] with the original specification (see Example 3.3). In other colors, we also include the syntactic and semantic similarities between the manually reported repairs (ground-truth) and the original specification for other cases than the MPC, in which possible solutions were proposed to remove the conflicts from the specifications. We can make two observations. On the one hand, some of the (manually developed) repairs produce small chances in the syntax of the goals but largely affect their semantics. However, other repairs are designed to maintain the fundamental semantics of the specification, although they may necessitate significant alterations to the syntax.

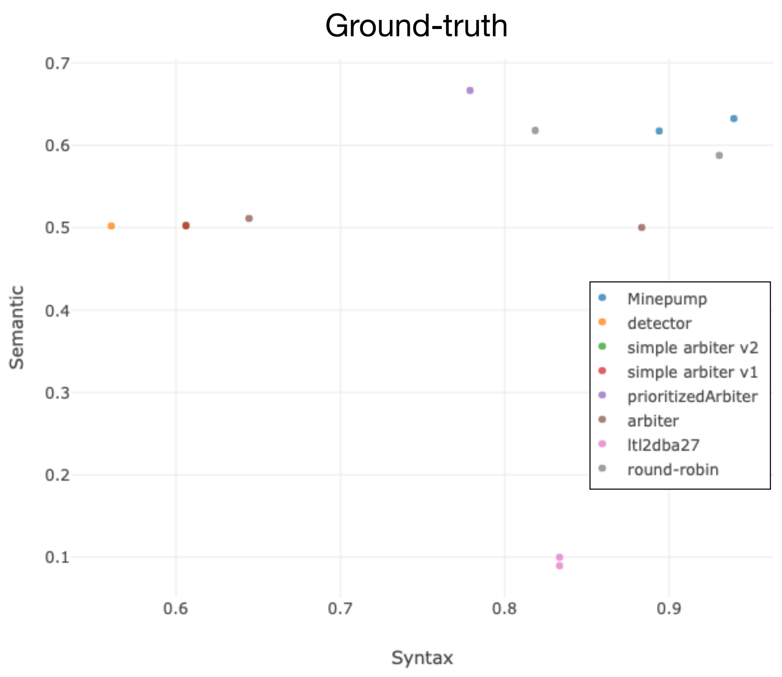


Figure 3.1: Syntactic and semantic similarities between the original specification and the manually developed repairs proposed by Letier [Let01] (ground-truth).

ACoRe was built on these observations and aims to explore candidate repairs that maximize syntactic similarity or semantic similarity to the original specification. Thus, ACoRe uses similarity metrics to guide a multi-objective search algorithm to produce (non-dominated) repairs that are (syntactically or semantically) similar to the original specification.

It is worth noticing that for this particular case, ACoRe is effective in generating repairs that exactly match the ground-truth cases. But let us discuss other interesting repairs generated by ACoRe that resolve the previously identified conflict (i.e., $BC = \Diamond(m \wedge h)$). These repairs were obtained during experimentation using the NSGA-III algorithm.

ACoRe attempts to produce repairs that do not contain the mentioned boundary condition, while preserving as much as possible behaviors not related to the boundary condition.

[Repair 1 - MPC] One of the repairs that can be produced by ACoRe is the following:

$$\begin{aligned} Dom :&\square((p \vee \bigcirc(p)) \rightarrow \bigcirc(\bigcirc(\neg h))) \\ G_1^1 :&\square(m \rightarrow \bigcirc(m \rightarrow \neg p)) \\ G_2^1 :&\square(h \rightarrow \bigcirc(\neg m \rightarrow p)) \end{aligned}$$

Repair 1 refines both goals G_1 and G_2, leading to G_1^1 and G_2^1, and thus conflict $BC = \diamond(m \wedge h)$ is resolved. In particular, the refined goal G_1^1 indicates that the pump should be off when methane is present in the environment and is still present in the next unit. On the other hand, G_2^1 indicates that if the level of water reaches a high threshold, then the pump should be on switch only if there is no methane present in the environment. Notice that in cases where a high level of water and methane is detected simultaneously, Repair 1 prefers to shut off the pump. In other words, the repair prioritizes the explosion condition, even though flooding is still possible.

[Repair 2 - MPC] Other repair produced by ACoRe is:

$$\begin{aligned} Dom :&\square((p \vee \bigcirc(p)) \rightarrow \bigcirc(\bigcirc(\neg h))) \\ G_1^2 :&\square(m \rightarrow \diamond(\neg p)) \\ G_2 :&\square(h \rightarrow \bigcirc(p)) \end{aligned}$$

Repair 2 preserves the second goal, and thus, the pump will be switched on mandatory in the presence of a high level of water. However, the first goal was to weaken and, in the presence of methane in the environment, it requires to switch off the pump at some point in the future, but not immediately after as it was the case in the original goal G_1. This will give the engineer some freedom to decide when the pump switch is the right time and satisfy the goal G_1^2. In contrast to Repair 1, this repair prioritizes flooding avoidance, while the explosion still can happen.

The two previous repairs modify the goals to add countermeasures (behaviors) that deal exactly with the identified conflicting situation. However, ACoRe produces less natural changes in goals to remove the boundary conditions. Hence, the repairs produced by ACoRe are validated by engineers.

[Repair 3 - MPC]

$$\begin{aligned}
Dom :& \Box((p \vee \bigcirc(p)) \rightarrow \bigcirc(\bigcirc(\neg h))) \\
G_1 :& \Box(m \rightarrow \bigcirc(\neg p)) \\
G_2^3 :& \Box(h \rightarrow \bigcirc(\Box(\neg p)))
\end{aligned}$$

This repair only modifies the second goal. The refinement goal G_2^3 states that once the high level of water is reached, the pump should go off forever. Despite the fact that this repair removes the boundary condition, it excessively changes the behavior goal G_2. Then, engineers can discard this repair if they consider that it is less natural or inappropriate to deal with the identified conflict.

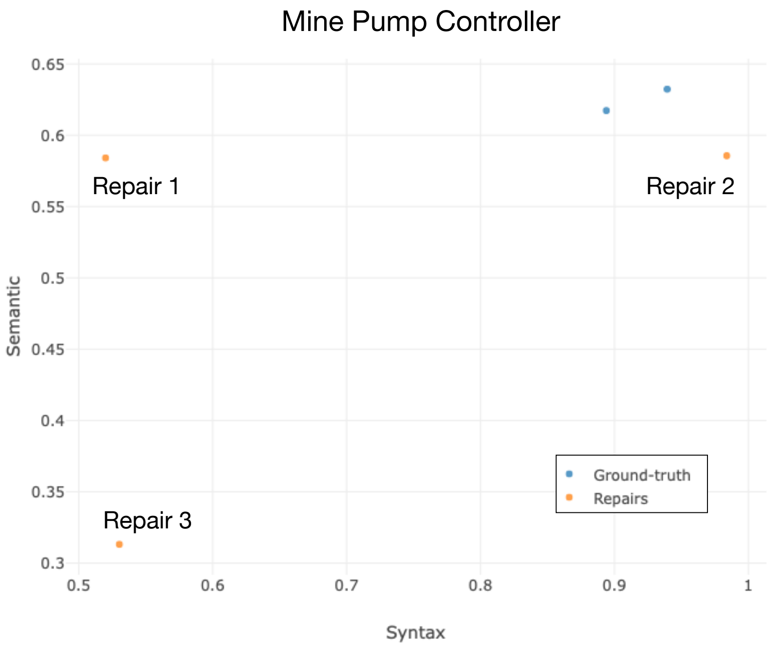


Figure 3.2: Mine Pump Controller: Syntactic and semantic similarities between the ground-truth repairs and the ones generated by ACoRe w.r.t. the original specification.

Figure 3.2 shows the syntactic and semantic similarities of the three mentioned repairs, w.r.t., to the original specification, and compares with the values of the ground truth repairs. While "Repair 1" obtains a semantic similarity value close to the value of the ground truth, it is syntactically more dissimilar to the original specification than the ground truths. In contrast, "Repair 2" obtains a syntactic similarity value greater than that of the ground truth, although a smaller semantic similarity is close to the ground truth values. In the case of "Repair 3", it is both syntactically and semantically more dissimilar to the original specification than to the ground truth.

Engineers can rely on additional requirements criteria and stakeholders' expertise to validate and select the repairs that are better suited for their purposes. Given our similarity metrics, we would prioritize Repair 1 and Repair 2, among Repair 3, since they maximize at least one of the similarity metrics. In general, Repair 1 and 2 are present in the Pareto optimal set computed by ACoRe, while Repair 3 is not.

3.4 ACoRe: Automated Goal-Conflict Resolution

ACoRe works on the specification $S = (Dom, G)$, in which Dom are domain properties and G are goal properties. Moreover, ACoRe also receives a set of previously identified boundary conditions BC for the specification mentioned S. Overall, ACoRe is a search-based software engineering approach. Thus, it integrates a search to explore variants of S with the aim of producing a set R of repairs that resolve all identified goal conflicts and maintain some sort of similarity to the original specification.

Figure 3.3 shows an overview of the different steps of the search process implemented by ACoRe. ACoRe instantiates multi-objective optimization (MOO) algorithms to explore the search space. We integrated four MOO algorithms, namely a *Weight-based genetic algorithm (WBGA)* [Hol92], an *Archived Multi-objective Simulated Annealing (AMOSA)* [BSM$^+$08] approach, the *Non-Dominated Sorting Genetic Algorithm III* (NSGA-III) [DJ14], and an *unguided search* approach we use as a baseline. In what follows, we describe the common components shared by the algorithms. For instance, the search space representation, finesses, and mutation operators.

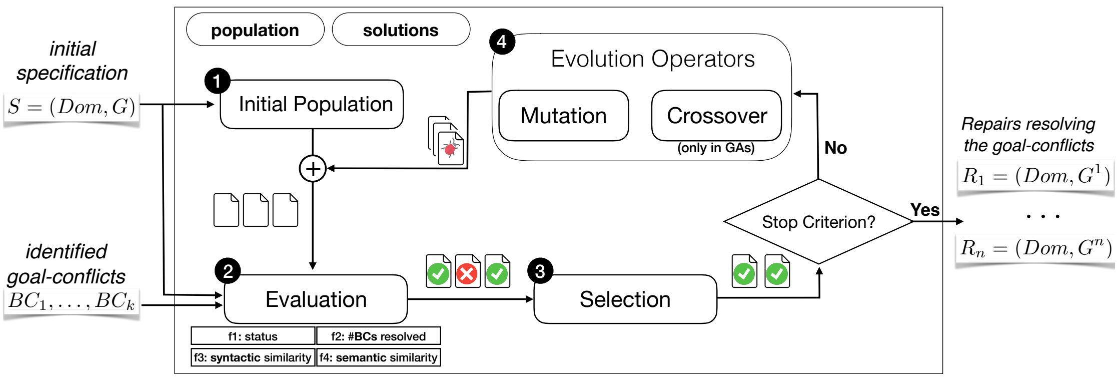


Figure 3.3: Overview of ACoRe.

3.4.1 Search Space

ACoRe performs syntactic alterations to the goals (prescriptive statements) to obtain new refined goals in G'. Thus, the search space contains candidate repairs $cR = (Dom, G')$ to resolve the goal-conflicts given as input. Each individual cR (candidate repair) is an LTL specification over a set AP of propositional variables, structured as a pair (Dom, G'), in which Dom represents the domain properties and G' the (refined) system goals. According to Definition 6, domain properties are not changed during the search process, as they are considered descriptive statements.

3.4.2 Initial Population

The initial population represents a sample of the search space from which the search process starts. ACoRe creates individuals as the initial population by applying the mutation operator (explained below) to the specification S given as input. Depending on the MOO algorithm being executed, ACoRe will create one or more variants of the initial specification.

3.4.3 Search Objectives

ACoRe guides the search with a multi-objective fitness function. Let us describe the different objectives that ACoRe takes into account and then present the details of the fitness function that guides the MOO algorithms.

Given a candidate repair $cR = (Dom, G')$, ACoRe considers four (4) objectives:

- *Consistency*(cR) computes if the refined goals are satisfiable and consistent with the domain properties.
- *ResolvedBCs*(cR) computes the ratio of boundary conditions from BC that are resolved by cR.
- *Syntactic*(S, cR) computes the syntactic similarity (cf. Definition 7) between the candidate repair cR and the original specification S given as input.
- *Semantic*(S, cR) computes the semantic similarity (cf. Definition 8) of the candidate repair cR w.r.t. the original specification S given as input.

We define $Consistency(cR)$ as follows:

$$Consistency(cR) = \begin{cases} 1 & \text{if } Dom \wedge G' \text{ is satisfiable} \\ 0.5 & \text{if } G' \text{ is satisfiable,} \\ & \text{however } Dom \wedge G' \text{ is unsatisfiable} \\ 0 & \text{if } G' \text{ is unsatisfiable} \end{cases}$$

$Status(cR)$ returns 1, whether the goal resolution found is satisfiable. Therefore, we claim that cR solves the conflict characterized by BC (Definition 6). $Status(cR)$ returns 0.5, whether the domains and goals properties are unsatisfied. In the case that G' is unsatisfiable, then it returns 0.

Similarly, *ResolvedBCs* computes the ratio of resolved boundary conditions by the total boundary conditions provided in the input.

ResolvedBCs(cR) is defined as follows:

$$\mathit{ResolvedBCs}(cR) = \frac{\sum_{i=1}^{k} \mathit{isResolved}(cR, BC_i)}{k}$$

The similarity measures are calculated by the syntactic and semantic similarities between the candidate repair and the original specification, denoted by *Syntactic*(S, cR) and *Semantic*(S, cR).

Similarity metrics are typically used by automatic program repair (APR) techniques [AGS19; JXZ+18; NQR+13; WCW+18], for example, to focus the search for candidate repairs in the vicinity of the program under analysis given as input. ACoRe will rely on two similarity metrics, namely *syntactic* and *semantic* similarities, to explore candidate repairs that are in some sense similar to the original specification given as input.

In particular, *syntactic similarity* refers to the distance between the text representations of the original specification and the candidate repair. To compute the syntactic similarity between LTL specifications, we use the Levenshtein distance [LY19]. Intuitively, 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.

Definition 7 (Syntactic Similarity)**.** *Let φ and ψ be two LTL formulas. The syntactic similarity between φ and ψ, denoted by Syntactic(φ, ψ), is calculated as:*

$$\mathit{Syntactic}(\varphi, \psi) = \frac{maxLength - lv(\varphi, \psi)}{maxLength}$$

where $maxLength = max(length(\varphi), length(\psi))$ and $lv(\varphi, \psi)$ represent the Levenshtein distance between φ and ψ.

Syntactic(φ, ψ) represents the ratio between the number of tokens changed from φ to obtain ψ $(lv(\varphi, \psi))$, among the maximum number of tokens corresponding to the largest specification.

Semantic similarity refers to the system *behavior* similarities described by the original specification and the candidate repair. Precisely, semantic similarity will be computed as the ratio between the number of behaviors present in both the original specification and candidate repairs among the number of all the behaviors described by the specifications.

Definition 8 (Semantic Similarity)**.** *Let φ and ψ be two LTL formulas. Given a bound k for the lasso traces, the semantic similarity between φ and ψ, denoted by*

Semantic(φ, ψ), *is computed as:*

$$Semantic(\varphi, \psi) = \frac{\#(\varphi \wedge \psi, k)}{\#(\varphi \vee \psi, k)}$$

To efficiently compute our semantic similarity metric, ACoRe uses model counting and, precisely, the approximation previously described in Definition 4.

Notice that small values for $Semantic(\varphi, \psi)$ indicate that the behaviors described by φ are very different from those described by ψ. In particular, when $\varphi \wedge \psi$ is unsatisfiable, $Semantic(\varphi, \psi)$ is 0. As this value gets closer to 1, both specifications characterize an increasingly large number of common behaviors.

3.4.4 Evolutionary Operators

Mutation

The offspring is created by the application of the mutation functions *mutate* as shown in Definition 16. ACoRe randomly selects a goal $G_i \in G$ that comes from a candidate individual $cR = (Dom, G)$ to mutate and produce a new candidate repair $cR' = (Dom, G')$. Given a set of goals G from individual cR, ACoRe invoke the function $mutate(G_i) = G'_i$ to produce a modification of the goal G_i, guiding it to the mutated goal G'_i. For example, *mutate* recursively replaces an unary operator, binary operator, or even an atomic proposition for others. In addition, it inserts or removes them. Thus, ACoRe creates a new individual $cR' = (Dom, G')$. This new individual will be assessed by the fitness function to analyze how fit it is to resolve the given goal-conflicts. ACoRe will use this mutation operator to produce variants of goal specifications with the aim of resolving goal-conflicts.

Definition 9 (LTL Mutation)**.** *Given an LTL formula* ϕ*, the function* $mutate(\phi) = \phi'$ *mutates* ϕ *by performing a syntactic modification driven by its syntax. Thus,* $mutate(\phi) = \phi'$ *is inductively defined as follows:*
Base Cases:

1. *if* $\phi = true$*, then* $\phi' = false$*; otherwise,* $\phi' = true$*.*

2. *if* $\phi = p$*, then* $\phi' = q$*, where* $p, q \in AP$ *and* $p \neq q$*.*

Inductive Cases:

3. *if* $\phi = o_1\phi_1$*, where* $o_1 \in \{\neg, \bigcirc, \Diamond, \Box\}$*, then:*

   (a) $\phi' = o'_1\phi_1$*, s.t.* $o'_1 \in \{\neg, \bigcirc, \Diamond, \Box\}$ *and* $o_1 \neq o'_1$*.*

   (b) $\phi' = \phi_1$*.*

   (c) $\phi' = o_1 mutate(\phi_1)$*.*

(d) $\phi' = p\ o_2'\ \phi$, *where* $p \in AP$ *and* $o_2' \in \{\mathcal{U}, \mathcal{W}, \wedge, \vee\}$

4. *if* $\phi = \phi_1 o_2 \phi_2$, *where* $o_2 \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$, *then:*

   (a) $\phi' = \phi_1\ o_2'\ \phi_2$, *where* $o_2' \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$ *and* $o_2 \neq o_2'$.

   (b) $\phi' = \phi_i$, *s.t.* $\psi_i \in \{\phi_1, \phi_2\}$

   (c) $\phi' = mutate(\phi_1)\ o_2\ \phi_2$.

   (d) $\phi' = \phi_1\ o_2\ mutate(\phi_2)$.

General Cases:

5. $\phi' = o_1\phi$ *where* $o_1 \in \{\Box, \Diamond, \bigcirc, \neg\}$.

6. $\phi' = x$, *where* $x \in \{true, false\} \cup AP$

Base cases 1 and 2 replace constants and propositions with other constants or propositions, respectively. Inductive case 3 mutates unary expressions: it can change the unary operator by other (3.a), remove the operator (3.b), mutate the sub-expression (3.b), or augment the current formula by including a binary operator and a proposition (3.d). Inductive case 4 mutates binary expressions: it can change the binary operator (4.a), remove one of the expressions and the operator (4.b), or mutate one of the sub-expressions (4.c and 4.d). Cases 5 and 6 are more general in that the entire formula ϕ is augmented with one unary operator (5) or replaced by a constant or proposition (6).

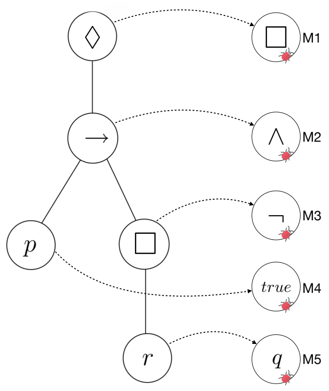


Figure 3.4: Examples of the mutations that ACoRe implements.

Figure 3.4 shows 5 possible mutations that can be generated for formula $\Diamond(p \to \Box r)$. Mutant M1 replaces $\Diamond$ by $\Box$, leading to $M1 : \Box(p \to \Box q)$. Mutant $M2 : F(p \wedge \Box r)$ replaced $\to$ by $\wedge$. Mutant $M3 : \Diamond(p \to \neg r)$ replaced $\Box$ by $\neg$. Mutant $M4 : \Diamond(true \to \Box r)$, reduced to $\Diamond\Box r$, replaced p by *true*. While mutant $M5 : \Diamond(p \to \Box q)$ replaced r by q.

Crossover

The two variants of genetic algorithms that ACoRe integrate, i.e. WBGA [Hol92] and NSGA-III [DJ14], besides the mutation operator, also consider a crossover operator to combine LTL goals from different candidates. To do so, they employ the *combine* function described in Definition 17.

Given two individuals $cR^1 = (Dom, G^1)$ and $cR^2 = (Dom, G^2)$, ACoRe first selects two goals such that $G^1_i \in G^1$ and $G^2_j \in G^2$, and then combines them to produce a new candidate repair $cR'' = (Dom, G'')$. Assuming that $G^1 = \{G^1_1, \ldots, G^1_{i-1}, G^1_i, G^1_{i+1}, \ldots G^1_m\}$ and $G^2 = \{G^2_1, \ldots, G^2_{j-1}, G^2_j, G^2_{j+1}, \ldots G^2_k\}$, then $G'' = \{G^1_1, \ldots, G^1_{i-1}, G''_i, G^1_{i+1}, \ldots G^1_m\}$ where $G''_i = combine(G^1_i, G^2_j)$.

Typically, genetic algorithms that manipulate LTL formulas also implement a crossover operator [DMR⁺18]. This operator takes two LTL formulas ϕ and ψ, and produces a new formula combining parts of both ϕ and ψ. This operator will later be used by the two variants of genetic algorithms that ACoRe integrates, namely a *Weight-based genetic algorithm (WBGA)* [Hol92] and the Non-Dominated Sorting Genetic Algorithm III (NSGA-III) [DJ14].

Definition 10 (LTL Crossover)**.** *Let ϕ and ψ be two LTL formulas. Function combine(ϕ, ψ) will produce the new formula ϕ' by performing the following steps:*

1. *It selects a sub-formula α from formula ϕ; to be combined;*
2. *it selects sub-formula β from ψ;*
3. *it either, (a) $\phi' = \phi[\alpha \mapsto \beta]$ replaces α by β in ϕ; or (b) $\phi' = \phi[\alpha \mapsto \alpha\ o_2\ \beta]$ combines α and β with a binary operator $o_2 \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$.*

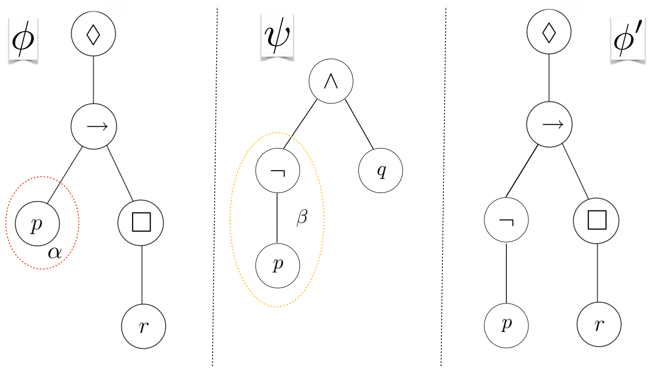


Figure 3.5: Examples of the combinations that ACoRe implements.

Figure 3.5 shows an overview of how this operator works in case a (i.e., option 3.a). In this example $\phi : \Diamond(p \rightarrow \Box r)$ and $\psi : \neg p \wedge q$. The selected formulas are $\alpha : p$ and $\beta : \neg p$. Thus, $combine(\phi, \psi)$ returns $\phi' : \Diamond(\neg p \rightarrow \Box r)$.

3.5 Research Questions

We start our analysis by investigating the effectiveness of ACoRe for resolving goal-conflicts. Thus, we ask:

RQ1 *How effective is ACoRe at resolving goal-conflicts?*

To answer RQ1, we evaluate the capability of ACoRe to produce repairs. In that way, we evaluate the three algorithms and the unguided search in terms of the number of repairs in the Pareto-optimal set. First, we evaluate the capability to find repairs. Second, we identify the causes of algorithms that do not produce them. Finally, we discuss the general number of repairs.

RQ2 *How capable is ACoRe to generate resolutions that match with resolutions provided by engineers (i.e., manually developed)?*

Regarding RQ2, we compare the logical equivalence between the repairs produced by ACoRe and the ground-truth. In the case that they are not equivalent, we compute the mean distance to the Pareto-optimal front in terms of hypervolume and inverted generational distance (IGD).

RQ3 *Do the repairs generated by ACoRe introduce new goal-conflicts?*

RQ3 extends RQ1 once we evaluate whether ACoRe is capable of producing at least one repair that does not introduce a new boundary condition. Although the approach is incomplete and cannot guarantee the absence of conflicts, it is up to date the most effective approach for identifying and repairing goal-conflicts.

RQ4 *What are the quality indicators of ACoRe when adopting different search algorithms?*

Finally, we compute standard quality indicators (e.g., hyper-volume (HV) and inverted generational distance (IGD)) to compare the Pareto-optimal sets produced by ACoRe when the different search algorithms are employed.

3.6 Experimental Setup

3.6.1 Subjects

We collected a total of 25 requirements specifications from different benchmarks. These specifications were previously used by goal-conflicts identification and assessment approaches [AMT13; DCA$^+$18; DMR$^+$18; DRA$^+$16; LWS$^+$21; vLDL98a]. Table 5.2 presents the specifications used in our empirical evaluation. For each case, the table depicts the number of domain properties and goals and the number of boundary conditions (i.e., goal-conflicts) computed with the approach of Degiovanni *et al.* [DMR+18]. Since the goal-conflict identification approach in [DMR+18] implements a genetic algorithm, there are many random decisions that can lead to a different set and number of boundary conditions. Thus, the number reported is the number of weakest boundary conditions of 10 runs, which are later used to run ACoRe. We use the set of "weakest"[1] boundary conditions returned by [DMR+18], in the sense that by removing all of them we are guaranteed to remove all the boundary conditions computed.

Table 3.1: LTL Requirements Specifications and Goal-conflicts Identified.

Specification	*#Dom + #Goals*	*#BCs*
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
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

3.6.2 Experimental Procedure

First, we run the approach of Degiovanni *et al.* [DMR+18] on each subject to identify a set of boundary conditions. Then, we run ACoRe to generate repairs

[1]A formula A is weaker than B, if $B \land \neg A$ is not satisfactory, i.e. if B implies A.

that remove all the identified goal-conflicts. We set as a termination criterion 1000 individuals as the maximum number of individuals explored by the different algorithms. We repeat this process 10 times to reduce the potential threats [AB11] raised by the random elections of the search algorithms implemented by ACoRe. Moreover, we also set two days as a time threshold per entire execution that includes the search process and identification of new boundary conditions. After that, we run ACoRe with the boundary conditions, which means that we do not perform any prioritization (e.g., [DCA$^+$18; LWS$^+$21]). However, we consider the weakest set of boundary conditions for the 10 executions referred to.

After that, we run ACoRe by using the four implemented search algorithms and report the Pareto-optimal set of repairs produced by each one. We focus on the effectiveness of the different search processes to generate repairs that resolve the given goal-conflicts. That is, we answer RQ1. Regarding RQ2, we analyze whether ACoRe is able or not to reproduce manually developed repairs (ground-truth). We collected from the literature, 8 cases in which authors reported a "buggy" version of the specification and a "fixed" version of the same specification. We analyze the buggy version, and we ask ACoRe to make some repairs, then we compare the repairs made by our tool and the fixed version in two aspects.

In this process, we analyze, by using an SAT solver, if there is any equivalent solution to the ground-truth. Second, for the cases where the repairs are not equivalent to ground truth, we study their similarity by using the *hypervolume* and the *inverted generational distance* indicators. When answering RQ2-4, we consider the AMOSA, NSGA-III, and WBGA algorithms to be the same repairs in the Pareto-optimal set computed for RQ1. However, for the unguided search algorithm, we randomly select four repairs (mean of repairs found in the Pareto-optimal front) from the set of all reached repairs, since our goal is to use this algorithm as a baseline for the rest of the algorithms, and it was not guided by any objective (then we do not want to "guide" the selection with the fitness objectives neither).

To answer RQ3, we check if the repairs reported by ACoRe introduce some (new) conflict. That is, for each repair produced by ACoRe, we run the same approach of Degiovanni *et al.* [DMR$^+$18] to look for goal-conflicts on the repair. Notice that if the repair resolves some conflicts but introduces others, then it would not be very interesting for the engineer, and we may overfit the ability of ACoRe to find appropriate repairs in our evaluation. Precisely, we report the percentage of repairs (in the Pareto-optimal set) produced by ACoRe that do not introduce new conflicts (notice, however, that still may exist some conflict not identified by the approach of Degiovanni *et al.* [DMR$^+$18], since it uses search and thus it is incomplete).

To answer RQ4, we perform a quantitative comparison of the four search algorithms implemented by ACoRe. We compare the Pareto-optimal set produced by each algorithm by using two standard quality indicators: hypervolume (HV) [ZTL+03] and inverted generational distance (IGD) [CR04].

3.6.3 Quantitative Evaluation Metrics

To evaluate the quality of the Pareto-optimal sets computed by ACoRe (RQs 2 and 4), we compute two standard quality indicators for assessing Multi-Objective Optimization Algorithms: the hypervolume (HV) and inverted generational distance (IGD) as aforementioned in the RQ4.

Hypervolume (HV) [LY19; TI20] is a *volume-based indicator*, defined by the Nadir Point [LM19; ZTL+03], that returns a value between 0 and 1, where a value near to 1 indicates that the Pareto-optimal set converges very well to the reference point [LY19]. In addition, high values for HV are a good indicator for the uniformity and spread of the Pareto-optimal set [TI20]).

The spread computes how the Pareto-optimal set covers the entire reference point or set, while the uniformity means how the uniform is the distribution of elements in the Pareto-optimal set. The union of spread and uniformity is often called diversity. Moreover, the convergence is a criterion to represent how close the Pareto-set is to the reference point.

Thus, we also compute the Inverted Generational Distance (IGD). IGD measures the mean distance from each reference point to the nearest element in the Pareto-optimal set [CR04; TI20]. The indicator is a *distance-based indicator* that computes convergence and diversity [LY19; TI20]. Low values for IGD indicates a good Pareto-optimal Front.

We also employ some statistical analysis when answering RQ3 and RQ4. For that reason, we use the Mann-Whitney U-test [MW47], Kruskal-Wallis H-test [KW52], Wilcoxon signed-rank test ($p - value$) [Wil45] and Vargha-Delaney A measure $\hat{A}_{12}$ [VD00] in order to compare the performance of the different algorithms implemented by ACoRe. Intuitively, $p - value$ will tell us if the performance of the algorithms measured in terms of HV and IGD are statistically significant, while the A measure will tell us how often one algorithm obtains better indicators than the others.

3.6.4 Implementation

ACoRe is implemented in Java in the JMetal framework. [NDV15]. It also integrates the LTL satisfiability checker Polsat [LPZ+13], a portfolio tool that runs in parallel four LTL solvers, helping us to improve the performance of the SAT checks required by the fitness functions. Moreover, ACoRe uses the OwL library [KMS18] to parse and manipulate the LTL requirements specifications. Quality indicators are also implemented by the JMetal framework, and statistical tests by Apache Common Math.

The replication package is publicly available at `https://sites.google.com/`

`view/acore-2022`. It contains our current implementation, the dataset with the 25 specifications, the reported results, and a description of how to reproduce the experiments.

We ran all the experiments on a cluster. Each node of the cluster has two Xeon E5 2.4GHz, with 5 CPUs-nodes and 8GB of RAM available per running. The specifications were resolved in the cluster using one node for each run. We ran ACoRe ten (10) times per specification.

3.7 Experimental Results

3.7.1 RQ1: Effectiveness of ACoRe

Table 3.2: Effectiveness of ACoRe in producing repairs. Column "#repairs" represent the average number of repairs in the Pareto-optimal set produced by the search algorithms.

Specification	**NSGA-III** #repairs	**WBGA** #repairs	**AMOSA** #repairs	**Unguided** #repairs
minepump	5.0	**6.5**	1.8	5.1
simple arbiter-v1	**4.8**	3.1	2.0	4.1
simple arbiter-v2	3.1	**3.4**	2.3	0.5
prioritized arbiter	3.1	**3.7**	2.2	0.0
arbiter	**5.8**	2.7	3.0	5.5
detector	4.9	4.8	3.2	**6.1**
ltl2dba27	3.0	**4.2**	3.5	4.0
round robin	**7.0**	4.2	4.7	4.7
tcp	6.4	4.9	2.0	**7.4**
atm	3.9	**6.3**	3.3	4.5
telephone	**4.7**	4.4	2.2	4.5
elevator	**5.9**	**5.9**	3.6	4.8
rrcs	5.5	**5.7**	1.4	3.3
achieve pattern	5.0	**5.9**	2.5	2.8
retraction pattern-1	4.1	4.0	2.7	**4.6**
retraction pattern-2	**6.1**	4.8	2.6	6.0
RG2	3.3	**5.2**	1.5	4.3
lily01	**5.1**	**5.1**	1.5	4.1
lily02	2.4	**3.8**	1.9	1.9
lily11	**7.1**	5.0	2.2	5.8
lily15	**6.1**	4.1	1.2	5.8
lily16	**3.5**	3.2	0.8	3.8
ltl2dba theta-2	1.9	**2.8**	1.9	1.2
ltl2dba R-2	1.0	**2.1**	1.9	**2.1**
simple arbiter icse2018	**3.8**	3.7	0.9	3.5

Column "#repairs" in Table 3.2 reports the average number of repairs in the Pareto-optimal set produced by four algorithms. First, it is worth mentioning that ACoRe when using the genetic algorithms, either NSGA-III or WBGA, it successfully generates at least one repair for all case studies. However, AMOSA produces 0.8 repairs on average for the lily16 case and 0.9 for simple arbiter icse2018, meaning that it failed in 2 and 1 out of the 10 runs, respectively, in producing repairs that resolve the identified boundary conditions. Unguided search succeeded in the majority of the cases, but it failed in the prioritized arbiter (no repair generated in the 10 runs), and in the simple-arbiter-v2 produces 0.5 repairs (meaning that it failed at least in half of the 10 runs).

Second, the genetic algorithms (NSGA-III and WBGA) typically generate more (non-dominated) repairs than AMOSA and unguided search. More precisely, WBGA generates more (non-dominated) repairs than others in 13 out of the 25 cases, and NSGA-III is the one that produces more (non-dominated) repairs in 11 cases. Unguided search generates more repairs in 3 cases, namely, detector, tcp and retraction-pattern-1. Finally, NSGA-III and WBGA outperform AMOSA and unguided search in 21 cases and coincide in one case (ltl2dba R-2). Interestingly, the different algorithms of ACoRe produce a reasonable number of repairs in the Pareto-optimal set (between 1 and 8) that can easily be explored by the engineer to select and validate the most appropriate one.

ACoRe generates more non-dominated repairs when adopting genetic algorithms than AMOSA or unguided search. In general, the algorithms of ACoRe produce an acceptable number of repairs in the Pareto-optimal set (between 1 and 8) to present to the engineer for analysis.

3.7.2 RQ2: Comparison with the Ground-truth

For this evaluation, we focus on 8 cases, for which we found in the literature a buggy version of the specification (that we aim to repair) and a manually developed fix, which is used as the ground-truth.

Table 3.3: ACoRe effectiveness in producing a exact match with the ground-truth

Specification	**NSGA-III**	**WBGA**	**AMOSA**	**Unguided**
arbiter				
detector	✓	✓		✓
ltl2dba27				
minepump	✓	✓	✓	✓
prioritized arbiter				
round robin				
simple arbiter-v1				
simple arbiter-v2	✓	✓		

Table 3.3 presents the effectiveness of ACoRe in generating a repair that matches exactly with the ground-truth. Overall, ACoRe was able to reproduce the same repair as our ground truth in 3 out of 8 cases, namely, for the minepump (our running example), simple arbiter-v2, and detector. Again, both genetic algorithms perform better than AMOSA and unguided-search in this respect. In this manner, the unguided search can replicate the fix for the detector case in which AMOSA fails.

In the cases in which ACoRe failed to replicate the ground-truth, we compute how close are the generated repairs. In other words, we measured the similarity of the Pareto-optimal sets when the ground-truth is a reference point. Table 3.4 shows the closest repairs produced by ACoRe to the ground-truth, where "distance" is measured in terms of the hyper-volume (HV) and inverted generational distance (IGD) indicators. As we already mentioned, ACoRe generated a perfect match for the detector, minepump, and simple-arbiter-v1 cases.

In the additional five cases, when we consider the hyper-volume (HV) indicator, NSGA-III algorithm was able to produce the closest set for the simple-arbiter-v1 and arbiter cases; while WBGA algorithm obtained closer repairs in two cases (prioritized arbiter and ltl2dba27), while the AMOSA algorithm obtained the closest repairs in the round robin subject. In the case of the unguided search algorithm, it obtained the best hyper-volume value, the same as WBGA, in the ltl2dba27 case.

When we consider the inverted generational distance (IGD), we observe that one of the genetic algorithms (NSGA-III and WBGA) obtained better values in 5 out of the 8 cases. Moreover, AMOSA obtains better IGD values in 3 out of the 8 cases in our ground-truth.

Overall, the genetic algorithms NSGA-III and WBGA, are more effective than AMOSA and unguided search to produce repairs that exactly match with the ground-truth. Moreover, NSGA-III and WBGA are more effective than AMOSA and unguided search in producing repairs that are closer to the ground-truth w.r.t. hyper-volume and inverted generational distance metrics.

3.7.3 RQ3: New conflicts introduced by repairs

First, we analyze the effectiveness of the algorithms in producing *at least one repair* that do not introduce new goal-conflicts for each case study. We observe that AMOSA was the most effective in this respect in 24 out of the 25 case studies, while WBGA succeeded in 22 out of the 25 cases, NSGA-III in 21, and unguided search in 20.

We also analyze the ratio among the 10 executions per case study in which each algorithm successfully generates at least one repair without new conflicts. We observe that WBGA succeeded in 83.2% of the executions, NSGA-III in 74.3% of the executions, AMOSA in 73.3%, and Unguided search in 64.6%. We

Table 3.4: HV and IGD between the closest repairs produced by ACoRe to the ground-truth

Specification	**NSGA-III**		**WBGA**		**AMOSA**		**Unguided**	
	HV	*IGD*	*HV*	*IGD*	*HV*	*IGD*	*HV*	*IGD*
arbiter	**0.83**	0.03	0.51	**0.01**	0.62	0.02	0.52	0.03
detector	***0.51***	*0.68*	*0.49*	***0.23***	0.35	0.41	*0.50*	*0.41*
ltl2dba27	0.91	0.13	**0.92**	**0.05**	0.37	0.06	**0.92**	0.13
minepump	***0.67***	*0.01*	*0.65*	***0.00***	*0.55*	***0.00***	*0.63*	***0.00***
prioritized arbiter	0.81	**0.85**	**0.95**	1.09	0.79	**0.85**	0.00	1.00
round robin	0.81	0.08	0.72	0.06	**0.87**	**0.01**	0.70	0.08
simple arbiter-v1	**0.99**	0.85	0.51	0.97	0.78	**0.68**	0.51	1.02
simple arbiter-v2	***0.99***	*0.83*	*0.94*	*0.76*	0.78	**0.75**	0.99	1.11

observe a skewed distribution ratio in the NSGA-III, AMOSA, and unguided search algorithms. Despite of WBGA presents a boxplot comparatively more symmetrical, less distributed, and has a high-ratio of no new boundary conditions. Thus, we can not assume, in general, a normal or another known distribution.

Moreover, we conduct a comparison regarding the absolute number of introduced new boundary conditions in the Pareto-fronts. We compare the results in terms of non-parametric tests. First, the Kruskal–Wallis test by ranks determines a difference between the means of the four algorithms. Second, the Mann-Whitney U test presents the significance of one group distribution being greater or lower than the other. The difference between algorithms adopts the α value of 0.05, while the pairs comparison adopts the α value of 0.0084.

The comparison presents a significant overall difference between the four algorithms (p-value < 0.00001) by the Kruskal–Wallis test by ranks. The following analysis confirms that WBGA algorithms introduce fewer new boundary conditions since the p-values individually compared to NSGA-III and unguided are lower than 0.002. Moreover, the AMOSA algorithm introduces significantly lower new boundary conditions than unguided search (p-value < 0.00210). The last comparisons between the pair NSGA-III and AMOSA present no significant difference since the p-value is greater than 0.29470. In the same way, no significant differences are found in the NSGA-III comparison pair and the unguided search once the p-value is greater than 0.02323.

In essence, ACoRe introduces a few new boundary conditions (the ratio is from 64.6% to 83.2%). Among the algorithms, WBGA introduces less boundary conditions than other algorithms.

3.7.4 RQ4: Comparing the four Multi-objective Optimization Algorithms

We compare the Pareto-optimal set computed by the different approaches by using traditional quality indicators, namely, hyper-volume (HV) and inverted generational distance (IGD). Precisely, we compute the HV and IGD of the syntactic and semantic similarity values among the repairs in the Pareto-optimal sets. The reference point was the ground-truth. The reference point is the best possible value for each objective, which is 1. These will allow us to determine which algorithm converges the most to the reference point and produces more diverse and optimal resolutions.

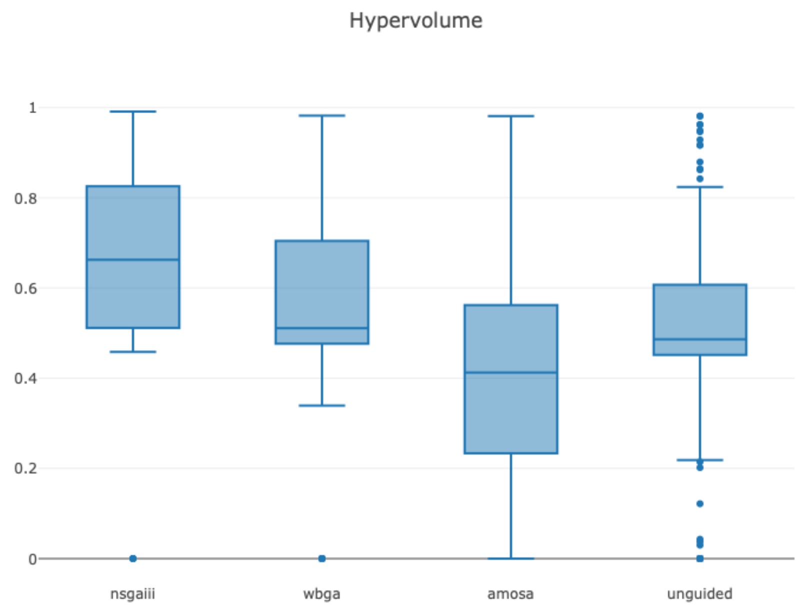


Figure 3.6: Boxplot that summarizes the HV of the Pareto-optimal sets generated by ACoRe when uses different search algorithm (higher value is the better).

Figure 3.6 and Figure 3.7 show the boxplots of the HV and IGD, respectively, computed under the Pareto-optimal sets when the four algorithms are adopted. The quality indicators show that NSGA-III algorithm obtains better performance than the rest. For instance, it obtains, on average 0.66, of HV (while higher the better) and 0.34 of IGD (while lower the better), outperforming the other algorithms. The NSGA-III box plots also show a low range between the quartiles and a better symmetry regarding medians and quartiles.

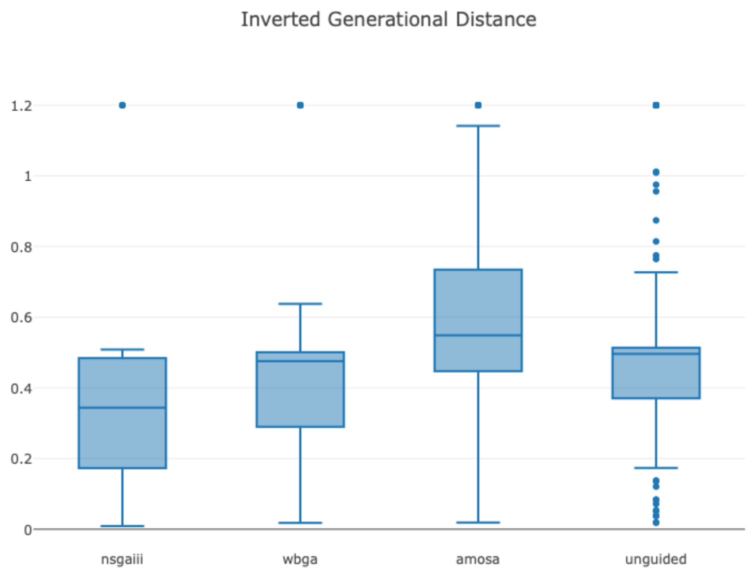


Figure 3.7: Boxplot that summarizes the IGD of the Pareto-optimal sets generated by ACoRe when uses different search algorithm (lower values is the better).

To confirm the differences and answer the RQ4, we compare the results in terms of non-parametric statistical tests: (i) Kruskal–Wallis test by ranks and (ii) the Mann-Whitney U-test. The α value defined in the Kruskal-Wallis test by ranks is 0.05, and the Mann-Whitney U-test is 0.0125. Moreover, we also complete our assessment by using Vargha and Delaney's $\hat{A}_{12}$, a non-parametric effect size measurement.

The Kruskal–Wallis test by ranks presents a p-value lower than 0.00001 for the hyper-volume indicator. The p-value is also lower than 0.00001 for the inverted generational distance. These results suggest a statistically significant difference between the four algorithms in terms of quality indicators.

We conduct the second statistical analysis to compare the algorithms and define the best algorithm in terms of the more optimized Pareto-optimal sets. In both indicators and pairwise comparisons, the NSGA-III shows a significant difference compared to the other three algorithms, leading to p-values lower than 0.00001. Moreover, the WBGA and unguided search algorithms are consistently better than the AMOSA algorithm (the p values related to both indicators are lower than 0.00001). Finally, WBGA performs better when compared to the unguided search (the p-value is also lower than 0.00001).

The Vargha and Delaney's $\hat{A}_{12}$ assessment also confirms that NSGA-III has greater HV and lower IGD when compared with the other three algorithms. Moreover, the results also point out that WBGA is better than AMOSA and unguided search. Finally, the unguided search surpasses the AMOSA results. Table 3.5 presents the summarized results of the hypothesis test and effect size.

Table 3.5: HV and IGD assessment by ACoRe

		WBGA		**AMOSA**		**Unguided**	
		HV	*IGD*	*HV*	*IGD*	*HV*	*IGD*
NSGAIII	*p-value*	< 0.00001	< 0.00001	< 0.00001	< 0.00001	< 0.00001	< 0.00001
	$\hat{A}_{12}$	0.66	0.65	0.84	0.83	0.80	0.76
WBGA	*p-value*	-	-	< 0.00001	< 0.00001	< 0.00001	< 0.00001
	$\hat{A}_{12}$	-	-	0.74	0.74	0.64	0.61
AMOSA	*p-value*	-	-	-	-	< 0.00001	< 0.00001
	$\hat{A}_{12}$	-	-	-	-	0.36	0.36

Overall, both statistical tests evidence that NSGA-III leads to Pareto-optimal sets with better quality indicators. Moreover, WBGA is significantly better than unguided search and AMOSA, while AMOSA has poor performance, which is worse than unguided search in several cases.

The quantitative comparison based on standard quality indicators presents that NSGA-III outperforms the other algorithms. In other words, it produces Pareto-optimal sets with statistically significant higher HV and lower IGD.

3.8 Threat to Validity

Threats to external validity may be related to the case studies and specifications we used in our evaluation. To mitigate this threat, we searched for independent specifications collected from the literature and benchmarks, which are typically used for evaluation requirements analysis tools. Although results may not generalize to other specifications and domains (for instance, when a different specification language than LTL is employed).

Threats to internal validity are related to the implementation steps. Regarding the implementation of our approach, which includes third-party libraries, we mitigate this risk through several repetitions of various experiments and the adoption of reliable libraries. The multi-objective algorithms contain an elevated degree of randomness in their search process. For instance, the application of the evolutionary operators is completely dependent of random decisions. To mitigate this threat, we repeat our experiments 10 times, following traditional research guidelines of how random algorithms should be evaluated [AB11], to reduce potential threats raised by the random elections of the search algorithms.

Our assessment metrics, number of repairs in the Pareto-optimal set, number of repairs resembling the ground-truth and not introducing new conflicts, are intuitive and reflect the effectiveness of ACoRe. Moreover, quality indicators (such as the hypervolume (HV) and inverted generational distance (IGD)) are a standard way to compare multi-objective search algorithms. We also employ several non-parametric statistical tests (namely, Kruskal–Wallis H-test and Vargha and Delaney's $\hat{A}_{12}$) to assess if the algorithm performs significantly different each other.

Although the threats mentioned are, we are confident that the results will remain in other case studies. ACoRe can successfully produce several repairs that require the validation by part of a domain expert. In this work, we did not focus on the "readability" of the repairs or the comprehension from the point of view of engineers, but we use the syntactic similarity metric to measure these properties. All the mentioned limitations present an interesting direction that we hope to follow in future work.

3.9 Conclusion

Goal-conflict resolution is a key step in Goal-Oriented Requirements Engineering (GORE). Frequently, GORE methodologies use a logical formalism to specify the system-to-be behavior. For instance, the KAOS method uses linear temporal logic (LTL) for specification purposes. Moreover, LTL is used to specify reactive systems [AIL+07; GMR24; HP85]. Once the specification is given, LTL allows us to identify conflicts easier and find their resolution. The specification in which a known conflict was removed is often called repair. In this paper, we present ACoRe, the first automated approach to goal-conflict resolution. In general, ACoRe takes a goal specification and a set of conflicts previously identified, expressed in LTL, and computes a set of repairs that remove such conflicts. ACoRe is a search-based approach that considers many objectives. Among them, ACoRe has objectives that reduce the syntax distance and increase the semantic similarity between the original specification and the proposed repairs.

To evaluate and implement ACoRe, we adopted three multi-objective algorithms (NSGA-III, AMOSA, and WBGA) that simultaneously optimize and deal with the trade-off among the objectives. We evaluated ACoRe in 25 specifications that were written in LTL and extracted from the related literature. The evaluation showed that the genetic algorithms (NSGA-III and WBGA) typically generate more (non-dominated) repairs than AMOSA and an unguided search, which we implemented as a baseline in our evaluation. Moreover, the algorithms generate a reasonable number of repairs per specification (between 1 and 8), allowing the engineer to analyze and select the most appropriate repair. We also observed that, in general, the genetic algorithms (NSGA-III and WBGA) outperform AMOSA and Unguided Search in terms of several evaluations: number of repairs, number of repairs not introducing new conflicts, number of repairs resembling manually written fixes, and standard quality indicators (HV and IGD) for multi-objective algorithms.

The future work follows two fold. First, we explore the capability of boundary conditions to produce complex inputs and also find bugs even in the LTL solvers. In this hypothesis, we adapt the search-based process to a fuzzing engine, while integrating the reduction of boundary conditions, and semantic and syntax distance. Furthermore, we also test the solvers based on the specifications and repair candidates produced in the search process that works, such as seeds in the fuzzing perspective. The soundness of the LTL solvers is evaluated by the differential outputs of the solvers set. Second, we intend to extend ACoRe in the set of features. Among them, we intend to allow the extension of different fitness, such as the model counting heuristic and the syntax metric adopted. Moreover, the multi-objective algorithms are also extensible, as mentioned and evaluated in the paper. To extend, we intend to allow additional multi-objective algorithms implemented in the JMetal framework to turn the parameters and data structures (e.g., archive) into a hotspot. Among the advantages of extensibility and modifiability, they allow engineers to tune the parameters and data structures, as well as evaluate additional algorithms and evolutionary operators to specific cases.

4

Evaluating Specification Mining in Linear Temporal Logic

Linear temporal logic (LTL) is a powerful logical formalism that includes descriptive operators for future events that deal with concurrent behaviors. LTL has a general set of operators that impose high expressiveness and complexity. The formalism includes safety, liveness, and fairness properties. LTL is commonly used as a specification language for distributed, concurrent, and hardware systems. On the basis of the high level of expressiveness, tools have been created to mine LTL properties based on trace artifacts. A trace describes the relevant events that occur or the behavior changes of concurrent systems. The LTL mining tools analyze patterns or constraints in these traces and propose LTL properties that are valid in the provided traces. However, there is a gap in the evaluation of the capability of mining tools. For instance, if the tools are able to propose correct properties or reach a soundness set of LTL properties. Therefore, we propose an evaluation of LTL mining tools. First, we assess them based on the capability to find a ground truth set of LTL specifications. Second, we observe the capability to cover the ground truth set of LTL specifications. Our evaluation shows that the mining tools do not find LTL properties in a ground truth composed of formal specifications. Moreover, the experiment also points out that the mining tools have low precision in the semantic coverage of the ground truth

Contents

4.1 Introduction 57
4.2 Background 60
4.3 Pattern Match 62
4.4 Constraint Solver 63
4.5 Experiment 64
4.6 Research Questions 65
4.7 Evaluation 67

4.8 Threat to Validity 70
4.9 Conclusion 71

5

§

4.1 Introduction

Linear Temporal Logic (LTL) is a widely used logical formalism to specify reactive systems [MP92], which the systems have to react to an environment which cannot wait. For instance, operating system, control systems, interactive systems. The formalism includes descriptive operators for future events that deal with concurrent behaviors [Pnu77].

The formalism includes safety, liveness, and fairness properties [OL82]. The safety properties ensure that the behavior of the system is always within an acceptable range of finite behaviors in which no adverse events occur. In summary, nothing bad happens. In the case of liveness properties, it provides guarantees that the system eventually reaches some good set of states, whereas fairness properties are insensitive to addition or deletion of prefixes [KV99; Sis94]. The mentioned properties are able to describe the absence of common bugs in concurrent systems such as race condition and deadlocks.

Moreover, the operators impose a high level of expressiveness and complexity [SC85; BSS$^+$07]. Furthermore, the expressiveness of a system may be constrained by factors such as GR(1) as outlined by Piterman *et al.* [PPS06] or well-established specification patterns as discussed in Dwyer *et al.* [DAC99].

Furthermore, LTL enables verifiability, as it offers a formalism for defining correctness properties for state transition systems. That is, model-checking tools employ LTL as a formalism for the formal verification of the safety-critical system [Roz11]. Among them, we mention TLA+, Spin, and NuSMV. They adopts LTL and additional specification language or model languages for distributed and concurrency systems [Lam02; CCG$^+$02; Hol97]. Further applications of the intersection of model checking and LTL include the verification of hardware [Gup93].

Moreover, LTL is also a common formalism in formal requirements. In this context, goal-oriented requirements engineering (GORE) methodologies [DvLF93; vLam09], offer an accessible approach to modelling and analysing the objectives of the proposed system. In these approaches, requirements are organized around the notion of goals, which are prescriptive statements that specify the functions and behaviors that the system to be developed should do. Furthermore, they are assessed in a multitude of ways, including feasibility assessments and evaluations of potential threats or risks [CvL15; NSG$^+$18; DCA$^+$18]. The initial set of domain properties and the elicited goals are formally expressed in LTL, as are the sets of identified goal conflicts [vLam09].

LTL presents a multitude of applications in model checking, formal requirements, reactive systems, and in general, concurrent systems. Thus, different approaches and tools have been proposed to deal with LTL [LMS20; MR21; GMR24]. In that manner, a recent interest topic about LTL is the mining of specification from common software artifacts [LPB15; GNR$^+$21].

The mining problem has been instantiated by different approaches with some level of variable components. Essentially, the mining problems concerns LTL formulas mining from given artifacts. The common artifacts in this context are finite traces that represent a program or software by events captured by monitors at runtime. Moreover, the mining tools also deal with templates that represent potential structure of formulas and their instances can represent the events in the finite trace. The aforementioned structure of formulas is composed of LTL and propositional operators with wildcards in the location of variables.

Among the mining tools, we can mention two types of them. The pattern match and constraint solver. The pattern match approach receives a template for the mined LTL formulas. Based on this template, they generate a syntatical tree that represents the propositional and LTL operators. The wildcards are replaced by events in the finite trace, also provided by the users. The pattern match of new variables in the fixed structure provided by the templates of operators refers to a pattern match when it is iteratively verified about their validity in the finite trace.

The constraint solver transforms the mining problem into a partial weighted maximum satisfiability problem. In other words, the mining problem is transformed into a set of constraints in which an off-the-shelf solver is applied to identify a solution. Moreover, the approaches are also capable of converting a finite trace of events into constraints that are part of the partial weighted maximum satisfiability problem. The finite traces are positive or negative traces. It means the mined LTL properties are valid in the positive traces and the negative trace does not represent the mined LTL properties.

The Texada approach is a pattern match approach that analyzes LTL templates and converts them into a syntactic tree comprising atomic propositions at the leaves and operators at additional nodes. Texada traverses this syntactic tree to ascertain the validity of an LTL instance for a specific template. Texada considers a template expressed as an LTL property and provided by users. Additionally, Texada accepts traces as input [LPB15].

Sample2LTL is a representative constraint solver for the mining of LTL. As previously stated, it transforms the mining problem into a variant of the maximum satisfiability problem [Bie09]. Furthermore, the encoding of the maximum satisfiability problem employs the appropriate weights assigned to the various clauses. Subsequently, the search is conducted by an off-the-shelf solver for assignments to the formulas that maximize the total weight of the satisfied clauses. In that manner, a weight function is applied to each clause of a propositional formula with the objective of identifying a solution that maximizes the weight function. In the case of Sample2LTL as proposed by Gaglione *et al.*, the Z3 solver was integrated [GNR$^+$21].

Sample2LTL also accepts finite traces and property templates. Moreover, it also works with incorrect events in the trace. Based on that possibility, the approach works with a tolerance level of noise given by the user. Moreover, the constraint solver also accepts negative traces. They describe trace that the events are not accepted by the generated LTL specification.

The current empirical experiments about LTL mining problems relies on evaluations of time performance and reduction of properties number. Texada is evaluated in terms of three differing dimensions: (i) number of traces, (ii) trace length, and (iii) number of unique events. They vary the three dimensions and compare the time performance. Also, Texada experiment evaluates internal components and potential configurations. Moreover, Texada is evaluated in terms of two web pages and one concurrent program [LPB15]. However, the ground truth are not provided in advance and it is not directed compared with the LTL mining properties. For the case of Sample2LTL, the experiment also limited to comparison of time performance, timeouts occurrences, and properties size.

Our experiment relies on formal specifications written in LTL. They are used to generate finite traces and templates. In addition, formal specifications are also used as a ground-truth set. That is, it is expected that LTL mining tools are able to find at least part of the formal specifications, when they receive the associated finite traces and templates. From a practical point of view, our experiment compares the formal specifications that composes the ground-truth with the mined specifications. The comparsion occurs in terms of syntactic equivalence or if the LTL mined properties are able to implicate in the LTL properties of the ground-truth. Syntactic equivalence expects to find the same LTL property in the ground-truth, while the implication comparison instantiate a cover notions. That is, whether the mined LTL property is able to cover a LTL property in the ground-truth.

The result show that Texada is able to identify a large set of properties. Despite of it does not matche with a comparible original set of LTL specification extracted from the literature and used to generate finite traces provided as input. In the case of Sample2LTL, the tool is able to generate LTL properties for a few cases. In general, the mining approaches are unable to reproduce the original set of LTL properties.

Besides that, we examine the capabilities of Texada and Sample2LTL in mining LTL formulas from finite traces, covering an original set of specification properties written in LTL. Thus, we assess the precision of the mined properties in relation to the covered original properties. To instantiate this concept in LTL, we compute the weakest equivalence. This entails computing the implications of each mined property to to each original property. The results suggest that the mined properties are capable of covering the original properties in a few cases. In other words, the precision for the Texada and Sample2LTL are low. This low precision makes the active a time-consuming and error-prone process.

Section 6.2 introduces the background about LTL, mining problem, and automaton generation. Section 4.3 presents the pattern match approach, while Section 4.4 discusses the constrain solvers. Section 4.5 shows our experiment and Section 4.6 the research questions. The evaluation is in Section 4.7. Moreover, Section 2.2 shows the related work. Finally, Section 6.9 presents the conclusion.

4.2 Background

4.2.1 Linear Temporal Logic

LTL formulas are inductively defined using the standard logical connectives, and the temporal operators to describe future events. For instance, the boolean connective ($\vee$) and the traditional definitions for *true* and *false*. For future events, we commonly define next ($\bigcirc$) and until ($\mathcal{U}$). For a formula φ and position $i \geq 0$, we say that φ holds at position i of σ. Thus, we write $\sigma, i \models \varphi$. Thus, LTL formulas are defined as follows [CHV+18]:

Definition 11 (LTL Syntax)**.** *Let AP be a set of propositional variables:*

1. *constants true and false are LTL formulas;*
2. *every $p \in AP$ is a LTL formula;*
3. *$p \in AP$ and $\sigma, i \models p$ iff $p \in \sigma_i$;*
4. *$\sigma, i \models \neg\varphi$ iff $\sigma, i \not\models \varphi$;*
5. *$\sigma, i \models \varphi \vee \phi$ iff $\sigma, i \models \varphi$ or $\sigma, i \models \phi$;*
6. *$\sigma, i \models \bigcirc\varphi$ iff $\sigma, i+1 \models \varphi$;*
7. *$\sigma, i \models \varphi\mathcal{U}\phi$ iff there exists $n > i$ such that $\sigma, n \models \phi$ and $\sigma, m \models \varphi$ for all m, $i \leq m < n$;*
8. *if φ and ϕ are LTL formulas, then are also LTL formulas $\neg\varphi$, $\varphi \vee \phi$, $\bigcirc\varphi$, and $\varphi\mathcal{U}\phi$.*

We also consider other typical connectives and operators, such as, $\wedge$, $\Box$ (always), $\Diamond$ (eventually) and $\mathcal{W}$ (weak-until), that are defined in terms of the basic ones. That is, $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$, $\Diamond\phi \equiv \mathit{true}\,\mathcal{U}\ \phi$, $\Box\phi \equiv \neg\Diamond\neg\phi$, and $\phi\mathcal{W}\psi \equiv (\Box\phi) \vee (\phi\mathcal{U}\psi)$.

The models of LTL formulas are infinite traces, it is often the case that analysis is restricted to a class of canonical *finite* representation of *infinite* traces, such as lasso traces or tree models.

Definition 12 (Lasso Trace)**.** *A lasso trace σ is of the form $\sigma = s_0 \dots\ s_i(s_{i+1} \dots s_k)^\omega$, where the states $s_0 \dots s_k$ conform the* base *of the trace, and the loop from state s_k to state s_{i+1} is the part of the trace that is repeated infinitely many times.*

For example, an LTL formula $\Box(p \vee q)$ is satisfiable, and one satisfying lasso trace is $\sigma_1 = \{p\}; \{p, q\}^\omega$, where in first state p holds, and from the second state both p and p are valid forever. Notice that the base in the lasso trace σ_1 is the sequence containing only state $\{p\}$, while the state $\{p, q\}$ is the sequence in the loop part.

4.2.2 Mining Problem

We introduce the concept of LTL mining problem and additional concepts. First, we define the trace, that is, a sequence of symbols of the form $\sigma = s_0\ s_1 \dots$, where each s_i is a propositional valuation on 2^{AP} (i.e., $\sigma \in 2^{AP^\omega}$). The finite trace has the form $\sigma = s_0\ s_1 \dots s_n$. The length of a finite trace is given by $|\sigma|$ or $n + 1$.

The mining problem considers a finite trace and commonly a set of LTL properties with wildcards instead of propositional variables that represents a formula template, in which the mined LTL formula should follow the structure of LTL operators and instantiate the propositional variables.

Many different formats are available. For example, Texada considers a set of finite events split by an end-of-event, while Sample2LTT receives as input the status of variables per each s_i, where $0 \leq i \leq n$.

4.2.3 Linear Temporal Logic to Automaton

In formal methods, the model checkings need to translate a provided LTL formula to an automaton [Roz11]. They also need to be equivalent. In other words, the provided LTL formula and the translated automaton need to recognize the same ω-language. For that purpose, several algorithms have been proposed to translate an LTL formula to an automaton [RV07].

Traditional translation algorithms are based on tableau to translate from LTL to automata. Gerth [GDP$^+$95] *et al.* presents an easy-to-describe tableau-based approach. The tableau-based construction builds a graph that defines the states and transitions of the generalized Büchi automaton. The nodes are generated by decomposing LTL formulas by their boolean structure and by expanding the temporal operators to split into two parts.

The first represents the immediately true, while the second part has to be true for the next state. For example, the equivalence $\mu\mathcal{U}\Psi \equiv \Psi \vee (\mu \wedge (\bigcirc(\mu\mathcal{U}\Psi)))$ is used for these expansions. After that, the tableau-based search is followed by a depth-first search (DFS). The second part mentioned is especially processed in DFS. In other words, when the complete expansion of the current node is finished, then the expansion of the another node is started.

In fact, the main challenge of the translation process is that a translate algorithm ensures that some run of the generated automaton does not violate the semantics of the temporal operators. For that aiming, they build an automata that when reading a word, then it needs to monitor the residual part that persists to be satisfied. Thus, the transition functions explore the expansion laws for LTL to decompose the formula into parts that can be monitored on the current status and the residual parts to be monitored in the future [EKS20].

4.3 Pattern Match

Texada considers a template written as an LTL property and given by users. Moreover, Texada also considers as input traces that represent a program or software. The output generated by Texada is a set of LTL formulas that are instantiations of the previous provided templates. In fact, they are in accordance with the traces provided as inputs. That is, they are valid LTL formulas for the provided traces [LPB15].

Texada analyzes the LTL templates and converts to a tree structure whose leaves are atomic propositions, and the additional nodes are LTL operators. After that, Texada searches this syntactic tree to check the validity of an LTL instance for a particular template. Among the advantages of the syntactice tree representation, it allows to re-use evaluation results and also implements memoize algorithms. This is adopted because instantiations will commonly have identical sub-formulas and sub-tree. Thus, Texada uses the syntactic tree to implement the memoize and reuse algorithms. Once the subtree of two distinct template instances may be identical, the memoization process speed up the verification of similar LTL instantiations.

In regard to the Texada analyses, the traces can be transformed into two distinct representations. The initial representation is a linear array, whereas the subsequent representation is a map structure in which the events are mapped to a sorted list of trace locations. Finally, Texada searches for the validity of each template instantiation on events in the trace.

A natural approach to evaluating a property instance on a linear trace is to perform a recursive traversal of the trace, evaluating each operator in accordance with its semantics. As previously stated, Texada represents the LTL instantiations as a tree and checks the candidate by traversing the syntactic tree. For example, to check a potential conjunction operator, Texada traverses its children on the first event of the trace.

At the conclusion of the trace, attention is paid to the verification of the traces, ensuring guarantees with finite trace semantics. The majority of operators possess a straightforward boolean base case for the terminal event. Among them, the X-operator necessitates a more comprehensive base, creating the appearance of an infinite trace. This evaluation reaches a conclusion because the formula tree for any p is finite, then Texada matches an event at a leaf node.

An additional feature in Texada is the map trace representation that is able to skip over the trace between selected events. This feature makes the map checker algorithm more efficient than the linear algorithm. Nevertheless, the map checker algorithms have the same evaluation as the linear algorithm.

4.4 Constraint Solver

Constraint solver approaches transform the mining LTL formulas into a variant of maximum satisfiability problem (MAX-SAT) [Bie09], after which an off-the-shelf solver is employed to identify a solution. Furthermore, constraint solvers are capable of generating a decision tree on temporal formulas and Signal Temporal Logic (STL) [GNR$^+$21]. The generating process happens even in the presence of noise trace. That is, incorrect events in the trace. Based on that possibility, the approach works with a tolerance level of noise given by the user. Moreover, constraint solver also accepts negative traces. They describe trace that the events are not accepted by the generated LTL specification. In this work, we restrict the evaluation to LTL formulas with positive and negative traces.

In summary, the encoding of the maximum satisfiability problem also employs the appropriate weights assigned to the various clauses. Subsequently, the search is conducted by a off-the-shelf solver for assignments to the formulas that maximize the total weight of the satisfied clauses. Moreover, a loss function is defined which assigns a real value to a given sample and an candidate LTL formula. The loss function evaluates the degree to which the LTL formula covers the sample.

MAX-SAT is a variant of the boolean satisfiability problem. It aims to identify an assignment that maximizes the number of satisfied clauses in a given propositional formula. For mining LTL formula, Gaglione *et al.* [GNR$^+$21] adopts a common variant of MAX-SAT, known as Partial Weighted MAX-SAT. In that manner, a weight function is applied to each clause in the clauses of a propositional formula, with the goal of finding a solution that maximizes the weight function. The MAX-SAT and the partial weighted MAX-SAT may be solved by SMT solvers. In the case of Sample2LTL proposed by Gaglione *et al.*, Z3 solver was integrated [GNR$^+$21].

Overall, the constraints formulas encoded by Partial Weighted MAX-SAT should describe linear temporal logic semantics and the events in the finite trace. The constraints formulas are split into two large groups, the first, structural constraints, and second, semantic constraints.

Structural constraints are based on the syntactic representation of LTL formulas, which are essentially syntax Directed Acyclic Graphs (DAGs). In other words, the generated syntax tree never forms a closed loop. The syntax tree allows for the sharing of common subformulas, resulting in the number of unique subformulas of an LTL formula coinciding with the number of nodes.

To guarantee the validity of the syntax DAG encoding, it is essential to ensure that each node in the syntax DAG is uniquely labeled. Furthermore, it is necessary to guarantee that each node in the syntax DAG has a unique left and right child. Finally, the structural constraints are obtained by the conjunction of all the aforementioned structural constraints.

Semantic constraints aims to definition of the formula, in that manner, it is defined propositional formulas for each trace that tracks the valuation of the LTL formula encoded. The formulas are built by variables that correspond to the value of the LTL formula rooted at node i. To guarantee that the mentioned variables have the expected meaning, the approach constraints based on the semantics of the LTL operators. For instance, it enforces the semantic constraints of LTL future operators such as the X-operator. The final semantic constraints formula is the conjunction of all such semantic constraints. The semantic constraints guarantee that the expected LTL formula is evaluated in terms of trace and also according to the LTL semantics.

The complete mining algorithm comprises a loop that increases from 1 to n. This continues until the solver identifies an assignment that satisfies the specified constraints and ensures that the sum of weights is greater than those previously found. The algorithms terminate when a previously given value of n or an LTL formula with zero loss on the given trace sample is found.

4.5 Experiment

Our experiment starts with a set of LTL specification. Our LTL specification set contains a total of 24 LTL specifications collected from the literature. These formulas have previously been used by many approaches for the identification and resolution of divergences [AMT13; DCA$^+$18; DMR$^+$18; DRA$^+$16; vLDL98b; CDB$^+$23]. The aforementioned specifications are represented in terms of domain and goal properties, as indicated in the GORE methodology. [vLam09]. Figure 6.1 illustrates our experiment, while Table 5.2 summarizes the number of LTL formulas of each LTL specification.

Besides that, we translate all the LTL specification to a nondeterministic generalised Büchi automaton. In this step, we instantiate the translation proposed by Esparza *et al.* [EKS20]. In summary, the translation is achieved through a process of guessing and verifying the set of greatest fixed-point operators, such as always and weak-until, that are satisfied by the majority of suffixes in the word being read by the automaton. Additionally, the set of least fixed-point operators, such as eventually, is verified to be satisfied by an infinite number of suffixes in the word being read by the automaton. The implementation is avaliable in the OwL framework [KMS18].

In the next step, we perform a random walk in each automata. The random

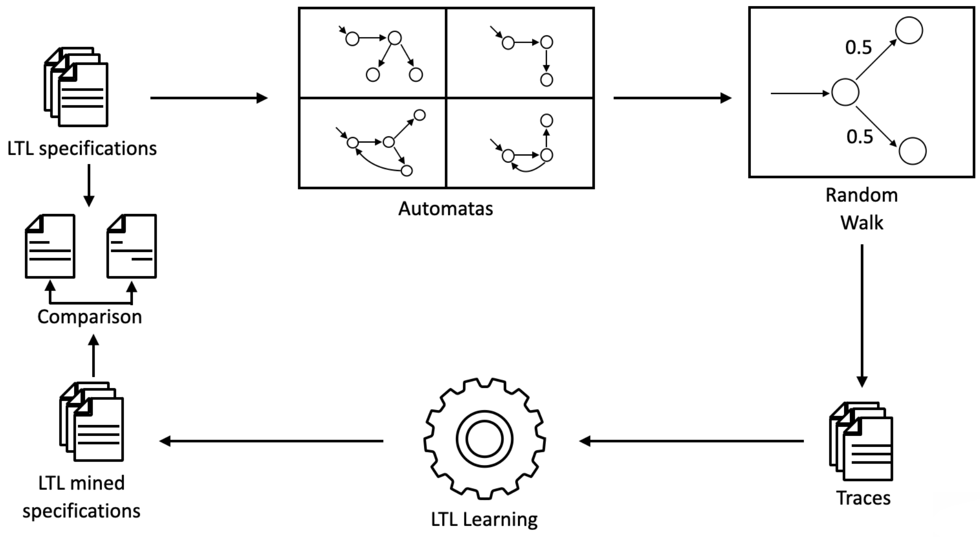


Figure 4.1: Comparison of mined specifications and ground truth of specifications

walk consists of randomly transverse each individual automata and save each state successor. The result of the random walk is a valid trace of an automata. Moreover, the random walk has a trace size threshold. After that, Figure 6.1 represents the LTL learning. In summary, the LTL learning step instantiate the LTL mining tools. In our experiment, we run Texada [LPB15] and Sample2LTL [GNR+21]. The result of this step is a LTL mined specification.

Finally, the mined specification is compared with the original LTL specification in the last step of our experiment. They are compared in terms of property existence of the LTL specification in the mined LTL specification. Additionally, we also compare the existence of the weakest equivalence of mined properties and the LTL specification.

4.6 Research Questions

We begin our analysis by examining the effectiveness of Texada and Sample2LTL in the LTL properties mining. We then ask:

RQ1 *How effective is LTL mining tools at identify LTL properties?*

Regarding RQ1, we evaluate the LTL mining tools regarding the matches of mined properties and the original LTL properties. Therefore, we start our experiment with a set of 25 LTL specifications in domain and goal properties. After that, we convert them into automatons and generate traces based on a random walk. The LTL mining tools received the generated traces. In this manner, they are

Table 4.1: Seeded LTL formulas and boundary conditions.

Specification	$\#\mathcal{S}$
minepump	3
simple arbiter-v1	4
simple arbiter-v2	4
prioritized arbiter	7
arbiter	3
detector	2
ltl2dba27	1
round robin	9
tcp	2
atm	3
telephone	5
elevator	2

Specification	$\#\mathcal{S}$
rrcs	4
achieve-avoid pattern	3
retraction pattern-1	2
retraction pattern-2	2
RG2	2
lily01	3
lily02	3
lily11	3
lily15	3
lily16	6
ltl2dba theta-2	1
ltl2dba R-2	1
simple arbiter icse2018	11

dissociated from the original LTL specification. The excepted results are for finding matches of the mined LTL properties that are also in the original LTL properties that come from the LTL specifications. The result suggests the effectiveness of LTL mining tools in the correct mining of LTL properties.

RQ2 *How capable is LTL mining tools to produces LTL properties that covers a original LTL specification?*

To answer RQ2, we again consider the set of mined LTL properties and the set of original LTL properties. We also compare both sets of properties. However, we adopt a cover notion instead of the precise matches of properties. For that reason, we use the weakest equivalence described by a simple implication. That is, a mined LTL property should be able to implicate at least an single original LTL property. In this manner, it is expected that the mined LTL property should semantically cover at least a single LTL property.

4.6.1 Experimental Instruments

We selected two LTL mining tools. The selected mining tools are state-of-the-art in different terms such as the theoretical approach adopted and the performance. In what follows, we describe the mining tools.

Texada Texadas represents a pattern match insofar as the main algorithm compares potential properties to scraps of the finite trace, iteratively. In our experiment, we provide 25 finite traces extracted by random walk into the generated automaton from the original set of LTL specification presented in Table 5.2. The templates correspond to the original properties when the variables are replaced by wild-cards.

Sample2LTL represents a constraint solver approach. The main reason is convert the LTL mining problem into a MAX-SAT problem. In that manner, the LTL syntax and semantics is converted into logical constraints. Moreover, future operators are also expressed as logical constraints. Also, the association with the traces and the templates provided as the input. Templates are patterns extracted from LTL specifications shown in Table 5.2 and the variable are replaced by wild-cards. Moreover, Sample2LTL is able to represent positive and negative finite traces. Thus, we provided 13 positive finite traces and 12 negative finite traces, both extracted by random walk into the generated automaton from the original set of LTL specification.

4.6.2 Experimental Setting

We ran Texadas and Sample2LTL experiments on a core-i7. Each node has 6 CPUs nodes available and 8GB of memory. The operating system is Linux, version 5.10. Moreover, we also set five hours as the time threshold for each execution. Our preliminary experiments show that an increase in the time threshold does not change the reported timeouts. In total, we executed 24 jobs, one job per each LTL specification in Table 5.2.

4.7 Evaluation

Table 4.2 presents the LTL specifications and their associated results produced by our experiment presented in Figure 6.1. Specifically, Table 4.2 is concentrated in the steps of mined specification and comparison.

4.7.1 Effective

To answer RQ1, we investigate the effectiveness of Texada and Sample2LTL in mining LTL formulas from finite traces. For the mined specification, we present the number of mined properties in the column properties of Table 4.2.

Texada generates LTL properties in 22 out of 24 cases. However, Texadas produces a large number of properties. The exception cases are Retraction pattern 1 and 2 in which Texada mined three properties and failed to find a single property, while for ltl2dba27, Texada is also unable to find properties.

We can compare the mined properties with the original set of LTL specifications and their properties shown in Table 5.2. The mean for the generated data is 178,128.375, while for the original LTL properties, the mean is 3.56. The mean for Texada mined specification is larges and shifted by cases such as the Simple

Table 4.2: Mined LTL properties and their matches and implications

Specification	**Texada**			**Sample2LTL**		
	Properties	Matches	Implications	Properties	Matches	Implications
Arbiter	54	0	4	TIMEOUT	TIMEOUT	TIMEOUT
Detector	3168	0	368	TIMEOUT	TIMEOUT	TIMEOUT
Lily02	831	0	0	TIMEOUT	TIMEOUT	TIMEOUT
Ltl2dba27	0	0	0	TIMEOUT	TIMEOUT	TIMEOUT
Minepump	153	0	56	1	0	0
Prioritized arbiter	ERROR	ERROR	ERROR	TIMEOUT	TIMEOUT	TIMEOUT
RG2	75	0	14	TIMEOUT	TIMEOUT	TIMEOUT
Round robin	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT	TIMEOUT
Rrcs	594	0	71	1	0	0
Simple arbiter v1	375789	0	7925	TIMEOUT	TIMEOUT	TIMEOUT
simple arbiter v2	315862	0	3431	TIMEOUT	TIMEOUT	TIMEOUT
Achieve pattern	52	0	2	TIMEOUT	TIMEOUT	TIMEOUT
Elevator	102	0	4	TIMEOUT	TIMEOUT	TIMEOUT
Lily01	297	0	120	TIMEOUT	TIMEOUT	TIMEOUT
Lily11	20	0	0	1	0	0
Lily15	966	0	178	1	0	0
lily16	5307	0	55	1	0	0
Ltl2dba R2	588	0	0	TIMEOUT	TIMEOUT	TIMEOUT
Ltl2dba theta 2	3570336	0	68	1	0	0
Retraction pattern 1	3	0	0	TIMEOUT	TIMEOUT	TIMEOUT
Retraction pattern 2	0	0	0	TIMEOUT	TIMEOUT	TIMEOUT
Simple Arbiter icse2018	456	0	116	TIMEOUT	TIMEOUT	TIMEOUT
tcp	94	0	0	TIMEOUT	TIMEOUT	TIMEOUT
telephone	334	0	26	TIMEOUT	TIMEOUT	TIMEOUT

arbiter v1, v2, and ltl2dba theta2. In that manner, we also compute the median for both cases. Texadas generates in median 225.0 properties, while the original set of LTL specfication contains in median 3 properties.

Sample2LTL is the opposite of Texada. It produces a few properties and commonly failure to find a single property. In summary, Sample2LTL finds properties in 6 out of 25 cases. The mean is 0.24 properties, while the timeout cases are reduced to zero properties. The median is 0.0 for Sample2LTL. In summary, Sample2LTL is less effective in finding LTL properties and often reaches the timeout for our experiment.

Furthermore, we compare the existence of original LTL properties in the mined LTL properties. For that analysis, we search for the precise match of the original LTL properties in the set of mined LTL properties. The collumn Matches of Table 4.2 presents that Texada and Sample2LTL are unable to find original LTL properties.

In summary, the result suggests that Texadas finds a large set of mined LTL properties that can cause big efforts to be analyzed by engineers, and Sample2LTL finds an extremely reduced set of properties. Moreover, the result also suggests that Texada and Sample2LTL are not effective in the mining of LTL properties, once they have not mined LTL properties that present e matches with the original LTL properties.

4.7.2 Capability

We investigate the capability of Texada and Sample2LTL in mining LTL formulas from finite traces and cover an original set of specification properties written in LTL. To answer RQ2, we compute the precision of the the aforementioned mined properties in terms of covered original properties.

To instantiate the cover notion in LTL, we adopt the weakest equivalence. For that, we compute the implications of the mined properties and the original properties. For example, for a single mined property named M, and for a single original property N, we inquire if N, then M. In summary, the result shows if the mined properties are able to partially cover the original properties. For the mined specification, we present the number of valid implication or weakest equivalence in the column implications of Table 4.2.

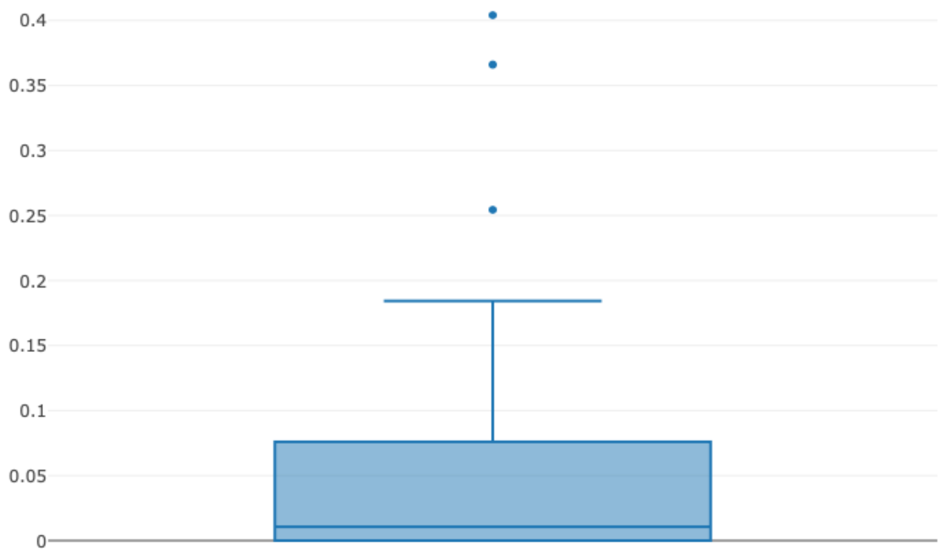


Figure 4.2: Box plot of precision for Texada.

Figure 4.2 presents the precision box plot. The median is 0.0106, while the minimum precision is 0.0 and the maximum is 0.4040. The errors and timeout cases have precision equal to 0.0. The precision box plot of Sample2LTL is absent since for all the cases, the precision is 0.0. In general, the precisions computed for the Texada and Sample2LTL tools have low ratios. From a practical point of view, developers need to analyze a large set of mined LTL properties to select and build a formal specification. The low precision makes it a time-consuming and error-prone active.

In general, the two presented research questions demonstrate that both tools are useless for obtaining formal specifications. The mining tools do not find the expected properties and are useless for partially covering the semantics described by the properties. Moreover, they produce a large number of properties (Texada) that it can turn the verification and validation process with high coasts into a manual analysis. In the case of Sample2LTL, a few properties are mined. However, the mined properties are not inserted in a semantic close context.

4.8 Threat to Validity

This section describes threats to the validity of the study.

Automaton translation to LTL is a potential source of threats of validity. In this manner, we used a state-of-the-art automaton translation [EKS20]. Moreover, the automaton translation is implemented by a standard LTL framework (OwL) [KMS18]. The empirical evaluation also compares the Esparza *et al.* [EKS20] algorithm against three of different algorithm types in terms of runtime performance and generated automaton size. Moreover, it consider the capability to generate automaton for different types of LTL formulas.

Moreover, we implement a random walk algorithm to produce finite traces. This algorithm contains a high level of randomness features. In this manner, we mitigate randomness bias by running the random walk algorithm several times and using the finite traces produced in our experiment. We have 25 finite traces per each LTL specification.

The automaton is rich in information and produces rich finite traces, whereas an existing program has a more limited or fewer number of variables in its finite trace. In this order, the representation of the finite trace can be problematic. For example, the representation does not consider all or many variables simultaneously. We do not deal with this threat and include the representation of finite trace as a capability of the LTL mining tools.

The automaton and its finite traces are rich in information. Compared to the existing software representation, the finite traces are usually extracted by injected monitors in the source code that capture a reduced subset of program behavior. Despite this rich environment created by the automaton, our experiment shows that the quality of LTL mining properties is low. We consider that for existing software. That is, we consider the result to be lower than the one present in our experiment in a constrained environment to produce finite traces. In the experiment, the decision supports the external validity mentioned.

The LTL specifications are a threat to validity. In order to mitigate this threat, we conducted a search for independent specifications that had been collected from the literature and benchmarks, which are typically used for the purposes of evaluation, requirements analysis, and tools analysis.

4.9 Conclusion

LTL has several applications in formal methods and concurrent systems. In order to easily specify a target system, a recent research topic about LTL is the mining of specification from common software artifacts. The mining problem has been partially handled by different approaches. Among them, pattern matches and constraint solver approaches. The LTL mining problems point out LTL formulas mining from given artifacts. The common artifacts are finite trace and templates to represent a target system.

The current empirical experiments about LTL mining problems relies on evaluations of time performance, reduction of properties number, timeouts occurrences, and properties size. Moreover, the experiments do not consider the ground truth in advance, and the ground truth is not directed compared with the LTL mining properties.

Thus, we present an experiment that relies on formal specifications written in LTL. These specifications are used to generate finite traces and templates through a random walk and automaton generation process. In addition, formal specifications are utilized as a ground truth set. This means that LTL mining tools are expected to be able to identify at least some of the formal specifications when they receive the associated finite traces. In conclusion, our experiment compares the formal specifications that comprise the ground truth with the mined specifications from the data set.

The results of the experiment indicate that there is no alignment between the LTL properties mined and a comparable original set of LTL specifications extracted from the literature and used to generate finite traces as input. In addition, we assess the capabilities of LTL mining tools, covering an original set of specification properties written in LTL. We evaluate the precision of the mined properties in relation to the covered original properties. The results indicate that the mined properties are capable of covering the original properties in a few cases. In summary, the precision of the LTL mining tools is very low. This low precision makes the active process time-consuming and error-prone.

5

Fuzzing Linear Temporal Logic Solvers

Fuzzing is a powerful technique to generate random input strings, while it can perform different types of tests and bugs may be found depending on the parameters provided and engines instantiated. Our work presents a new fuzzing technique to test Linear Temporal Logic (LTL) solvers. Our fuzzing technique is based on search-based fuzzing that generates and evolves failure-inducing input by evaluating fitness functions. Essentially, a boundary condition represents a conflict condition whose occurrence results in the loss of satisfaction of the goal's properties. We claim that the search process for resolving boundary conditions efficiently produces failure-inducing input. Moreover, additional fitness, such as semantic and syntactic similarities, can constrain the search space and maintain the seeds given by the specifications as representative of realistic LTL specifications. Finally, the algorithm computes the satisfiability of the modified specification. The result suggests that our fuzzing is significantly more efficient than probabilistic grammar fuzzing. Our empirical evaluation presents 16 bugs in the evaluated LTL solvers. In addition, we present performance warning patterns. The set of warning patterns has the potential to indicate performance improvements in a particular solver for releases, heuristics set, or theoretical guidelines for a specific type of LTL satisfiability algorithm.

Contents

5.1 Introduction **74**
5.2 Background **76**
5.3 Boundary Conditions **80**
5.4 SpecBCFuzz **82**
5.5 Empirical Evaluation **89**
5.6 Evaluation **94**
5.7 Threats to Validity **101**
5.8 Conclusion **102**

5.1 Introduction

Linear Temporal Logic (LTL) is a logical formalism widely used to specify reactive systems [MP92]. Moreover, Goal-Oriented Requirements Engineering (GORE) methodologies have adopted LTL to formally express requirements and have taken advantage of the powerful manual or automatic analysis techniques associated with LTL [vLam09]. The main goal is to improve the quality of the specifications. Furthermore, LTL solvers have been proposed to address or improve model checking tools [Bie21], software and hardware verification [Gup93; DKW08], and artificial intelligence reasoning [CTM^{+}17]. LTL solvers are built on the basis of different and complex search algorithms and data structures. The state-of-the-art presents performance-oriented benchmarks and methods to evaluate LTL solvers. However, software testing techniques that identify bugs are usually absent.

In fact, LTL solvers are utilized in the development of critical systems, and it is of utmost importance that they are both correct and reliable. For instance, solvers are employed as part of verification tools, and erroneous solver results can result in a significant portion of the program under analysis remaining unverified, which in turn increases the risk of failing to identify potential bugs and vulnerabilities. Therefore, we explore black-box testing techniques to identify performance warnings and bugs in the implementation of solvers.

There are testing approaches for SAT and SMT solvers (see, for example, [BLB10; BMB^{+}18]), the problem of testing LTL solvers remains largely unexplored. To the best of our knowledge, there is no approach that automatically tests LTL solvers beyond some set of benchmark formulas [SD11]. Consequently, there is a gap in methods for purposely testing LTL solvers.

We address this gap by the proposal of SpecBCFuzz, a fuzzing method for LTL solvers that is guided by the LTL semantics. The fundamental concept is to generate formulas that are likely to prompt the solvers to explore potential corner cases, a general principle that has been successfully applied in other testing domains, such as boundary value analysis. We claim that boundary conditions are suitable for finding failure-inducing input in LTL solvers because they include two important (semantic) properties: i) the conjunction of specification with boundary condition 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 an excellent potential to trigger bugs, as shown by our results.

Our findings indicate that all fitness functions of SpecBCFuzz are effective in identifying relevant failure-inducing inputs when we optimize and search by boundary conditions, syntactic and semantic similarity, and general satisfiability of the modified LTL specifications. Furthermore, our analysis demonstrates that SpecBCFuzz outperforms a conventional grammar-based fuzzer that was specifically designed and optimized for fuzzing LTL solvers.

In the course of this search process, we assert that our tool is capable of generating failure-inducing input, given that the verification of boundary conditions precipitates an increase in the logical inconsistency property, shifting the answer from unsatisfiable to satisfiable. In other words, we utilize LTL solvers to their full potential in terms of satisfiability. Furthermore, additional fitness criteria, such as semantic and syntactic similarities, can constrain the search space and maintain the seeds provided by the specifications as representative of realistic LTL specifications.

In general, the empirical evaluation identified 16 bugs in the LTL solvers evaluated, with five distinct categories of bugs identified in the qualitative analysis of the inconsistency patterns. The identified categories of bugs are as follows: (i) exceptions, (ii) crashes, (iii) flaky behavior, (iv) hang loops, and (v) wrong answers. In our approach, the bugs are found by the inconsistency patterns in the differential fuzzing results. Inconsistency patterns are sets of inconsistencies that result in the same observed symptoms. For example, a single difference in the outcomes occurs when a solver answers differently from all the others. In addition, the inconsistency pattern can be a complex pattern such that all tableau-based solvers respond differently. Moreover, each inconsistency pattern is commonly associated with a single bug, taking into account their behavioral observation.

Furthermore, our evaluation analyzes a warning pattern, a method of associating timeout warnings from one or more solvers. Warning patterns are beneficial for performance testing, as they demonstrate performance issues in a specific LTL solver, version, or even an implemented algorithm. Consequently, the set of warning patterns has the potential to indicate improvements in a particular solver for releases, heuristic sets, or theoretical guidelines for a particular type of satisfiability algorithm. Our empirical result presents 315 warning patterns found by SpecBCFuzz.

Moreover, we also compute the similarities of the warning pattern using a binary indicator. The empirical result suggests associations of different types of algorithms and their search strategies for LTL satisfiability. From a practical point of view, developers of LTL solvers can select hard-to-response LTL formulas. For example, developers of an LTL solver can include in their evaluation LTL formulas that are hard to response to BMC algorithms, or tableau and BMC at the same time.

The empirical results point to groups of similarity, and the groups correspond to three large groups of the satisfiability algorithm: (i) BDD, (iii) BMC, and (iii) tableau. The empirical result shows that a single solver can implement more than a single strategy to answer LTL satisfiability. For example, the similarity coefficient is able to cluster formulas hard-to-response for Black in BMC and tableau clustering. In fact, Black generates a tableau and navigates in the tableau with a BMC-like strategy, as confirmed in previous papers [GGM$^+$21b].

We also performed a comparative analysis of the empirical results generated by SpecBCFuzz to ascertain whether the fuzzing strategies themselves are capable of producing similar outcomes when we disable the fitness functions and the seeds. In this manner, the bias in the final result of the tool was evaluated using different subsets of fuzzing strategies. To this end, an unguided search was conducted in the presence of realistic specifications and their associated boundary conditions. Furthermore, the semantic and syntax fitness functions were disabled to assess the impact of boundary condition fitness in the presence of previously evaluated seeds.

Moreover, we conducted a comparison between SpecBCFuzz and a probabilistic grammar fuzzing that represents the potential baseline for fuzzing LTL solvers. The empirical evidence suggests that SpecBCFuzz with a seed-based strategy represented by realistic LTL specifications, optimized in a search process for syntax and semantic similarity, satisfiability of offspring, and a reduced score of boundary conditions performs better than a reduced set of fuzzing strategies. SpecBCFuzz also significantly recognizes more inconsistency and warning patterns than probabilistic grammar fuzzing.

Our work extends the previous work [CDC$^+$24] in at least three components. First, we consider and compare explicitly inconsistency patterns and their occurrence. Statistical inference introduces SpecBCFuzz as better than a reduced set of fuzzing strategies. Seconds, the search-based fuzzing instantiated by SpecBCFuzz also over-performs a baseline represented by probabilistic grammar fuzzing. Moreover, we introduce the notion of a warning pattern, in which SpecBCFuzz identifies more patterns than a baseline. We also compute the similarity of solvers and the type of solvers. The results show that developers can perform the test with respect to formulas that are hard to respond to a particular solver or an association of LTL satisfiability algorithms.

In what follows, Section 5.2 presents the background. Section 5.3 introduces boundary conditions, while Section 5.4 presents SpecBCFuzz. The empirical study is reported in Section 5.5 and the empirical results in Section 5.6. In addition, threats to validity are discussed in Section 5.7, and related work is discussed in Section 2.3. Finnaly, Section 5.8 presents the conclusion.

5.2 Background

Linear Temporal Logic (LTL) solvers have been proposed to address or improve model checking tools [Bie21], software and hardware verification [Gup93; DKW08], and artificial intelligence reasoning [CTM$^+$17]. As aforementioned, LTL solvers are built on the basis of different and complex search algorithms and data structures. The state-of-the-art presents performance-oriented benchmarks and methods to evaluate LTL solvers. However, software testing techniques that identify bugs are usually absent. In this order, we explore black-box testing techniques to identify warnings and bugs in the implementation of the solvers. Thus, in this section, we present formal concepts about LTL. Besides that, we also introduce LTL solvers with different satisfiability algorithms and fuzzing strategies.

5.2.1 Linear Temporal Logic (LTL)

LTL formulas are inductively defined using standard logical connectives and temporal operators to describe future events [CHV+18]. For instance, the boolean connective ($\vee$) and the traditional definitions for *true* and *false*. For future events, we commonly define next ($\bigcirc$) and until ($\mathcal{U}$). Once the mentioned operators are defined, we may also define future operators, such as always ($\Box$) and eventually ($\Diamond$). LTL formulas are interpreted over infinite traces, for a formula φ and position $i \geq 0$, we say that φ holds at position i of σ. Thus, we write $\sigma, i \models \varphi$. In other words, 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}. LTL formulas are defined as follows:

Definition 13 (LTL Syntax)**.** *Let AP be a set of propositional variables:*

(a) constants true and false are LTL formulas;

(b) every $p \in AP$ is a LTL formula;

(c) $p \in AP$ and $\sigma, i \models p$ iff $p \in \sigma_i$;

(d) $\sigma, i \models \neg\varphi$ iff $\sigma, i \not\models \varphi$;

(e) $\sigma, i \models \varphi \vee \phi$ iff $\sigma, i \models \varphi$ or $\sigma, i \models \phi$;

(f) $\sigma, i \models \bigcirc\varphi$ iff $\sigma, i+1 \models \varphi$;

(g) $\sigma, i \models \varphi\mathcal{U}\phi$ iff there exists $n > i$ such that $\sigma, n \models \phi$ and $\sigma, m \models \varphi$ for all m, $i \leq m < n$;

(h) if φ and ϕ are LTL formulas, then are also LTL formulas $\neg\varphi$, $\varphi \vee \phi$, $\bigcirc\varphi$, and $\varphi\mathcal{U}\phi$.

Intuitively, formulas without a temporal operator are evaluated in the first state of the trace. Formula $\bigcirc\varphi$ is true at position i, iff φ is true at position $i+1$. Formula $\varphi\mathcal{U}\ \psi$ is true in σ iff formula φ holds at every position until ψ holds. We also consider other typical connectives and operators, such as conjunction ($\wedge$), always ($\Box$), eventually ($\Diamond$) and weak-until ($\mathcal{W}$), that are defined in terms of the basic ones. That is, $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$, $\Diamond\phi \equiv$ *true* $\mathcal{U}\ \phi$, $\Box\phi \equiv \neg\Diamond\neg\phi$, and $\phi\mathcal{W}\psi \equiv (\Box\phi) \vee (\phi\mathcal{U}\psi)$.

Definition 14 (Satisfiability)**.** *An LTL formula φ is said* satisfiable *(SAT) iff there exists at least one trace satisfying φ.*

The notion behind a LTL model checking is that satisfiability requires finding a trace or model that satisfies a logical formula [CHV+18], while the LTL solver answer if a formula contains at least one model that satisfies the formula, that is, the satisfiable or SAT answer. Otherwise, the LTL solver should be unsatisfiable or answer UNSAT.

5.2.2 Solvers

LTL solvers have been proposed to address the satisfiability problem stated in Definition 19. In an effort to answer the satisfiability problem, the LTL solvers were designed around several algorithms and data structures. Among the ends of additional effort, developments also address performance improvement [SD11], diversity of temporal operators [GGM+21a], and expressiveness [Lam02]. We may highlight successful strategies to address satisfiability: (i) Bounded Model Checking (BMC), (ii) Binary Decision Diagram (BDD), (iii) Tableau, and (iv) Automata-Theoretic Approach.

Bounded Model Checking (BMC) explores the state space based on examining a potential path segment that was constrained at a given length (k). The key idea is to identify a counterexample. Consequently, the solvers translate a path segment to a propositional formula, and the counterexample exists iff the propositional formula is satisfiable. If the segment is not found, the procedure continues for k+1 [CBR+01; BCC+99b]. NuSMV [CCG+02] implements a BMC algorithm based on propositional satisfiability. In other words, NuSMV builds a propositional formula that is satisfiable if and only if there exists a path that reaches a state of interest.

Binary Decision Diagram (BDD) associates elementary formulas or subformulas obtained by an input LTL formula [CGH97; CGP+02a]. Each boolean formula is then converted to a binary decision diagram (BDD) structure. The BDD maintains a compressed form that deals with a potential state-space explosion. Also, the structure instantiates common operators over the compressed form as conjunction, disjunction, and negation in polynomial time. The BDD structure generated by the obtained boolean formulas often represents an implicit tableau that allows the solving procedure.

Tableau methods build a graph or tree shaped when the definition of a node is a set of formulas and the relations are defined by this set. Based on Tableau, the LTL solving technique traverses to find suitable models for an input LTL formula. One-pass and multi-pass tableaux methods have been proposed. Among them, we observe the Schwendimann's method that is implemented by PLTL [Sch98a], and the Reynolds' method [Rey16b] implemented by BLACK [GGM21; GGM+21b].

Automata-Theoretic Approach translates LTL formulas to automata. Thus, an LTL formula describes a language or trees over some alphabet [Var96]. For this purpose, many automata were proposed. Among them, the alternating automata are commonly used for LTL satisfiability since the automata generalize the notion of non-deterministic by generating several successor states and are exponentially more succinct [Var96; KVW00]. In summary, the satisfiability of an LTL formula is reduced to inquire to the generated automata for a nonemptiness, where the nonemptiness problem for alternating automata is decidable in exponential time [KVW00]. Additional heuristics are used. For example, Aalta (version 1) implements an on-the-fly search and obligation sets [LYP+14].

5.2.3 Fuzzing

Fuzzing, or Fuzz testing, is a powerful technique to generate random input strings [MFS90]. Moreover, fuzzing can perform different types of tests [ZWC+22] such as system and integration tests, open-box tests, or even semantic-oriented tests [PLS+19; SZ22]. Bugs may be found depending on the parameters provided and instantiated engines. Furthermore, fuzzing addresses target software that may concern non-functional requirements such as security [FME+20; BMA+22].

Traditional fuzzing is able to find bugs in syntax parse modules. However, they hardly ever find bugs in the deeper functionality [PLS+19]. For instance, the bugs in semantic modules are non-conformance to specifications. In this order, different engines and strategies have been proposed.

Among the successful fuzzing engines, American Fuzzy Lop (AFL) has become one of them for automated security bugs. Moreover, the AFL influenced many variants of engines and also strategies. For instance, AFLFast extends in grey-box fuzzing and represents coverage of the system under testing as a Markov chain [BPR19], AFLGo defines an inter-procedural measure of distance and search by an annealing simulation [BPN+17], and Zest implements a property-based notion [PLS+19; PLS19]. In what follows, we present different strategies to build a fuzzing approach.

Grammar Fuzzing strategies are adopted to test programs built in two big modules: parsing and semantic analysis. For instance, compilers and network protocols. Grammar fuzzes produce and constrain inputs to a language, commonly defined by a context-free-grammar. The main goal is to produce diverse sets of valid inputs. Thus, grammar fuzzes overcome the limitation of high rates of inputs that just show syntactic errors. In other words, the grammar fuzzes aim to increase the chance of producing inputs that reach the syntax logic of the target system.

Probabilistic Grammar Fuzzing increases the power of mentioned grammar fuzzes insofar as probabilities are assigned to individual grammar expansion. The probabilistic is attributed to controlling how many of each element should be produced. For instance, if terminals are not common, considering a specific domain, then they may receive a lower probability than common terminals. Additionally, probabilistic grammar fuzzing also allows for the control of the largeness of inputs since the probabilistic ratio of non-terminal expansion and terminals are assigned.

Seeds are bootstrap of bug finding process [HGM+21]. Good seeds are collected from a large number of cases or representative domain-specific scenarios when the semantics of seeds are close to a semantic module of the target system. For example, seed sets are built by crawling the internet or getting representative inputs provided by users of the target system [SZ22; NP21; MDL+22]. Seed strategies are associated with strategies such as mutation-based or search-based.

Mutation Fuzzing introduces small changes to execute additional behaviors in the target system. Preferentially, the mutation keeps the input statically valid or domain adherent [CWB15]. Thus, mutation fuzzing is commonly joined to strategies such as grammar or seed-based fuzzing, when the grammar or seed set provides a valid input, the mutation fuzzing then subsequently changes its based on mutation operators. Typically, mutation operators remove, insert and flip a single bit, character, or terminal symbol from a grammar [WCG+13; LS18].

Search-based Fuzzing generates inputs during a search process, while the inputs generated are progressively improved and evaluated by fitness functions. The search process is instantiated by a wide range of optimization algorithms that includes evolutionary and stochastic meta-heuristics [RJK+17]. The optimized fitness functions are defined in the domain of the target system. Among them, coverage and input structure validity are adopted [AZG24].

5.3 Boundary Conditions

Goal-Oriented Requirements Engineering (GORE) [vLam09] methodologies provide an intuitive way to model and analyze the objectives of the envisioned system. In these approaches, requirements are organized around the notion of goals and domain properties. Goals are prescriptive statements that specify what the software to be developed should do, while domains properties are descriptive statements that commonly describe a physical or normative law that the software to be should take into consideration.

In this manner, GORE methodologies employ a logical formalism to specify the expected system behavior. Several GORE methodologies have also adopted LTL to formally express requirements [vLam09] and take advantage of the powerful automatic analysis techniques associated with LTL to improve the quality of their specifications (e.g., to identify inconsistencies [DMR+18]). In other words, goals are the subject of several activities, including goal decomposition, refinement [DvLF93; vLam09; vLDL98a], and feasibility [NSG+18]. In this context, a *conflict* essentially represents a condition whose occurrence results in the loss of satisfaction of the goals, that is, the goals diverge [vLDL98b; vLL98]. Formally, it can be defined as follows.

Definition 15 (Goal Conflicts)**.** *Let $G = \{G_1, \ldots, G_n\}$ be a set of goals and Dom be a set of domain properties, all written in LTL. Goals in G are said to diverge if and only if there exists at least one Boundary Condition (BC), such that the following conditions hold:*

- *logical inconsistency:* $\{Dom, BC, \bigwedge_{1 \leq i \leq n} G_i\} \models false$
- *minimality: for each* $1 \leq i \leq n$, $\{Dom, BC, \bigwedge_{j \neq i} G_j\} \not\models false$
- *non-triviality:* $BC \neq \neg(G_1 \wedge \ldots \wedge G_n)$

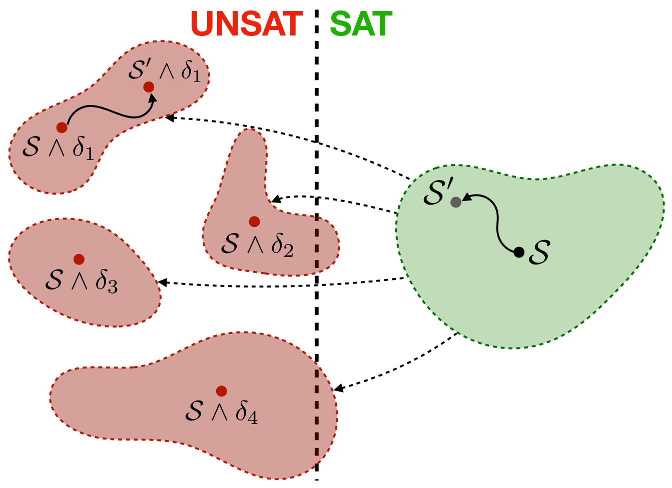
Intuitively, a BC captures a particular combination of circumstances with which the goals cannot be satisfied. The first condition establishes that, when BC holds, the conjunction of goals $\{G_1, \ldots, G_n\}$ becomes inconsistent. The second condition states that if any of the goals are disregarded, then consistency is recovered. The third condition prohibits a boundary condition from being simply a negation of the goals. In addition, the minimality condition prohibits BC to be equal to *false* (it must be consistent with the domain Dom).

From a GORE analysis perspective, the boundary conditions removal, once instantiated by a search process, captures a pragmatic perspective of continuous modification. The search process introduces changes in the goals specifications to take them from unfeasible specifications and boundary conditions to feasible repairs of the original buggy specification.

At the same time, from a software testing perspective, the LTL solvers receive formulas provided continually by the search process when the formulas are in the border of answering UNSAT and SAT insofar as the search process has as an objective breaks the logical inconsistency property. The additional properties just refer to the fact that the repairs are not trivial and useless "repairs" that completely have removed some objective (minimality) or negated all goals (non-triviality).

Overall, it considers a satisfiable formula (seed) $\mathcal{S}$ 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 generates a new candidate repair $\mathcal{S}'$ of 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 5.1.

Furthermore, goal specifications are also realistic LTL formulas produced by developers or researchers. They try to specify concurrent resources or systems. For example, LTL is a logical formalism widely used to specify arbitrators, schedulers, and even reactive systems [MP92]. Therefore, the properties contain safety, liveness, and fairness properties that also increase the complexity of the formulas. Thus, goal specifications rely on realists and complexes seeds for fuzzing LTL solvers.



UNSAT
SAT
$\mathcal{S}' \wedge \delta_1$
$\mathcal{S} \wedge \delta_1$
$\mathcal{S} \wedge \delta_2$
$\mathcal{S} \wedge \delta_3$
$\mathcal{S} \wedge \delta_4$
$\mathcal{S}'$
$\mathcal{S}$

Figure 5.1: Exploring the vicinity of the divergences.

5.4 SpecBCFuzz

SpecBCFuzz extends and adapts ACoRe [CDB+23]. ACoRe generates a set of repairs. They are consistent and free of any previously identified conflicts. For that aim, the tool employs search-based software engineering to efficiently explore syntactic variants of goals (throughout the application of evolutionary operators) to generate repairs that are similar to the original specification. In summary, the tool considers both syntactic and semantic similarity between the original specification and the candidate repairs, just as it implements a many-objective fitness function to direct the search.

Moreover, ACoRe also computes during the search process the number of boundary conditions fixed during the syntactic variants introduced and the satisfiability of the repair candidate. In other words, the repairs generated by the tool are those that are closest to a repair solution, which contains a reduced set of conflicts or even no conflicts.

In the current work, we extend and adapt ACoRe by SpecBCFuzz to interpret fitness objectives as fuzzing strategies and adapt them to generate failure-inducing inputs for LTL solvers. The section introduces components of the SpecBCFuzz as the many-objective algorithm. Moreover, this section also looks at the particular details of fuzzing strategies such that seeds and the evolutionary search are instantiated on top of fitness and operators that generate failure-inducing inputs.

Figure 5.2 presents SpecBCFuzz and their entire execution. Step 1 shows the evolutionary search, while step 2 calls the LTL solvers based on the complex LTL formulas produced during the search process. For example, the evolutionary search checks whether the initial boundary conditions are removed or not. Step 3 stores

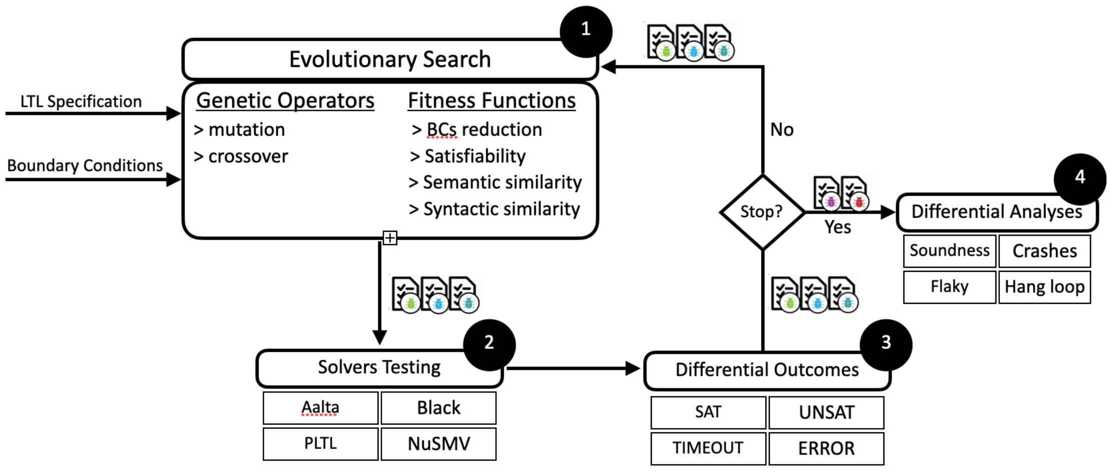


Figure 5.2: Evolutionary search implemented by SpecBCFuzz.

the different outcomes from LTL solvers. The last step occurs after the end of the evolutionary search, and the differential outcomes are analyzed by the presence of inconsistencies or warning patterns. For instance, the presence of errors and timeouts in the differential outcomes.

5.4.1 Evoluationary Search

Many software engineering problems contain more than one fitness function to be optimized [HJ01], and multi-objective algorithms optimize these problems. Essentially, those evolve the population to converge a set of solutions as close to the true Pareto front as possible. There are many multi-objective algorithms that have been successfully applied in practice [HMZ12].

The multi-objective evolutionary algorithms start by creating the initial population, composed of one or more individuals to be used to start the search. The algorithm evaluates and creates new individuals through fitness functions and the application of evolutionary operators to evolve and diversify the population. The mutation operator changes (mutates) a portion of the selected individual, while the crossover operator creates new individuals by combining parts of the other two individuals taken from the population.

Interactively, the algorithm then computes the fitness value for each objective taken into account for every new individual. It updates the solution set and selects the individuals that survive to the next iteration by discarding nonpromising individuals from the population set. This evolution process is performed until some termination criterion is reached, e.g., a defined number of generations, iterations, or a maximum number of individuals are created.

SpecBCFuzz integrates the Non-Dominated Sorting Genetic Algorithm III (NSGA-III) [DJ14] approach. It is a variant of a genetic algorithm. In each iteration, NSGA-III computes the fitness values for each individual and calculates the Pareto dominance relation between them. It uses this relation to create a kind of partition of the population in terms of the non-dominated level of the individuals (i.e., Level-1 contains non-dominated individuals, Level-2 contains the resulting non-dominated elements when all the individuals from Level-1 are not considered, and so on).

Thus, NSGA-III selects only one individual per non-dominated level with the aim of diversifying the exploration and reducing the number of solutions in the final Pareto-optimal set since the PO set can be huge, especially when multi-targets are used. This evolution process is performed until some termination criterion is reached, e.g., a defined number of generations, iterations, or a maximum number of individuals are created. SpecBCFuzz uses the NSGA-III algorithm until the individuals are explored and return the Pareto-optimal repairs that resolve all the boundary conditions given as input.

During this search process, we claim that SpecBCFuzz is capable of producing failure-inducing inputs since boundary condition verification jumps in the property of logical inconsistency from the UNSAT to the SAT answer and the opposite direction. That is, we call LTL solvers in the limits of satisfiability. Moreover, additional fitness, such as semantic and syntactic similarities, can constrain the search space and maintain the seeds given by the specifications as representative of realistic LTL formulas. Finally, the algorithm computes the satisfiability of the modified specification. The other sections present SpecBCFuzz in detail.

5.4.2 Search Based Fuzzing

SpecBCFuzz takes as seeds a set $S = (Dom, G)$ and their boundary conditions BCs, composed of the set of domain properties Dom, goals G, and boundary conditions BCs for the specification S. The input as fuzzing seeds represents realistic and complex LTL formulas since the specifications are commonly built by developers or researchers in complex scenarios.

The domain properties do not change through the evolutionary search process since they are considered descriptive statements. On the other hand, SpecBCFuzz uses search to iteratively explore variants of G to produce a set $R_1, \ldots, R_n$ of repairs that resolve all identified boundary conditions. Interactively, the algorithm computes the fitness value for each objective and takes it consideration for every new individual (cR). It then updates the solutions set and selects the individuals that survive to the next iteration by discarding non-promising individuals from the population set.

Therefore, the boundary conditions also as seeds strengthen the evolutionary search to produce new individuals that are repairs candidates on the border of feasible and infeasible specifications. In this manner, the LTL formulas are built based on the implication of domain properties to goals properties, satisfiability, and boundary conditions verification of the repairs. Thus, SpecBCFuzz produces complex formulas in the boundary of answering UNSAT and SAT for the LTL solvers.

Furthermore, the repair candidates maintain some sort of similarity with the original specification. Thus, SpecBCFuzz integrates two similarity metrics, and we claim that SpecBCFuzz considers one syntactic and one semantic similarity metric that will help the algorithms focus the search in the vicinity of the specification given as seeds.

5.4.3 Fitness

SpecBCFuzz guides the search with four objectives, which check the validity of consistency, resolution, and two similarity metrics.

Given a new individual $cR = (Dom, G')$, the first objective $Consistency(cR)$ evaluates if the refined goals G' are consistent with the domain properties by using LTL solvers.

$$Consistency(cR) = \begin{cases} 1 & \text{if } Dom \wedge G' \text{ is satisfiable} \\ 0.5 & \text{if } Dom \wedge G' \text{ is unsatisfiable, but } G' \text{ is satisfiable} \\ 0 & \text{if } G' \text{ is unsatisfiable} \end{cases}$$

The second objective $ResolvedBCs(cR)$ computes the ratio of boundary conditions resolved by the candidate resolution cR, among the total number of boundary conditions given as input. Hence, $ResolvedBCs(cR)$ returns values between 0 and 1, and is defined as follows:

$$ResolvedBCs(cR) = \frac{\sum_{i=1}^{k} isResolved(BC_i, G')}{k}$$

$isResolved(cR, BC_i)$ returns 1, if and only if G' breaks the property of logical inconsistency of BC_i as presented in Definition 15. Otherwise, returns 0. When $BC_i \wedge G'$ is satisfiable, it means that the refined goals G' satisfy the resolution condition. Thus, BC_i is no longer a conflict for the candidate resolution cR. In the case where cR resolves all the (k) boundary conditions, the objective $ResolvedBCs(cR)$ returns 1.

Specifically, objective $Syntactic(S, cR)$ refers to the distance between the text representations of the original specification S and the candidate resolution cR. We run the Levenshtein distance [Lev66] to compute the syntactic similarity between the LTL specifications and the candidate repair. In summary, 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, $Syntactic(S, cR)$, is computed as the ratio of the maximum length of an original or repair LTL specification:

$$syntactic(S, cR) = \frac{maxLength - Levenshtein(S, cR)}{maxLength}$$

where $maxLength = max(length(S), length(cR))$.

$Syntactic(S, cR)$ represents the ratio between the number of tokens changed from S to obtain cR among the maximum number of tokens corresponding to the largest specification.

On the other hand, our semantic similarity objective $Semantic(S, cR)$ refers to the system behavior similarities described by the original specification and the candidate resolution. $Semantic(S, cR)$ computes the ratio between the number of behaviors present in both the original specification and candidate resolution among the total number of behaviors described in the specifications. To efficiently compute the objective $Semantic(S, cR)$, SpecBCFuzz uses the model counting heuristic.

Despite existing approaches for computing the exact number of models being ineffective [FT14], Brizzio *et al.* [BCP+23] recently developed a novel and effective model counting heuristic that approximates the number (of prefixes) of lasso traces satisfying an LTL formula. Hence, given a bound k for the lasso traces, the semantic similarity between S and cR is computed as:

$$Semantic(S, cR) = \frac{\#\textsc{Approx}(S \wedge cR, k)}{\#\textsc{Approx}(S \vee cR, k)}$$

Small values for $Semantic(S, cR)$ indicate that the behaviors described by S are divergent from those described by cR. In particular, in cases where S and cR are contradictory (that is, $S \wedge cR$ is unsatisfiable), $Semantic(S, cR)$ is 0. As this value approaches 1, both specifications characterize an increasingly large number of common behaviors.

5.4.4 Evolutionary Operators and Selection

Mutation is described in the functions *mutate* in Definition 16. Given an individual $cR' = (Dom, G')$, the mutation operator selects a goal $g' \in G'$ to mutate, leading to a new goal g'', and produces a new individual specification $cR'' = (Dom, G'')$, where $G'' = G'[g' \mapsto g'']$ (i.e., G'' looks exactly like G' but the goal g' is replaced by the mutated goal g'').

Definition 16 (LTL Mutation)**.** *Given an LTL formula ϕ, the function $mutate(\phi)$ $= \phi'$ mutates ϕ by performing a syntactic modification driven by its syntax. Thus, $mutate(\phi) = \phi'$ is inductively defined as follows:*
Base Cases:

1. if $\phi = true$, then $\phi' = false$; otherwise, $\phi' = true$.

2. if $\phi = p$, then $\phi' = q$, where $p, q \in AP \wedge p \neq q$.

Inductive Cases:

3. if $\phi = o_1\phi_1$, where $o_1 \in \{\neg, \bigcirc, \Diamond, \Box\}$, then:

(a) $\phi' = o'_1\phi_1$, s.t. $o'_1 \in \{\neg, \bigcirc, \Diamond, \Box\}$ and $o_1 \neq o'_1$.

(b) $\phi' = \phi_1$.

(c) $\phi' = o_1 mutate(\phi_1)$.

(d) $\phi' = q\ o'_2\ \phi$, where $q \in AP$ and $o'_2 \in \{\mathcal{U}, \mathcal{W}, \wedge, \vee\}$

4. if $\phi = \phi_1 o_2 \phi_2$, where $o_2 \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$, then:

(a) $\phi' = \phi_1\ o'_2\ \phi_2$, where $o'_2 \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$ and $o_2 \neq o'_2$.

(b) $\phi' = \phi_i$, s.t. $\phi_i \in \{\phi_1, \phi_2\}$

(c) $\phi' = mutate(\phi_1)\ o_2\ \phi_2$.

(d) $\phi' = \phi_1\ o_2\ mutate(\phi_2)$.

General Cases:

5. $\phi' = o_1\phi$ where $o_1 \in \{\Box, \Diamond, \bigcirc, \neg\}$.

6. $\phi' = x$, where $x \in \{true, false\} \cup AP$

Base cases 1 and 2 replace constants and propositions with other constants or propositions, respectively. Inductive case 3 mutates unary expressions: it can change the unary operator by other (3.a), remove the operator (3.b), mutate the subexpression (3.b), or augment the current formula by including a binary operator and a proposition (3.d). Inductive case 4 mutates binary expressions: it can change the binary operator (4.a), remove one of the expressions and the operator (4.b), or mutate one of the subexpressions (4.c and 4.d). Cases 5 and 6 are more general insofar as the entire formula ϕ is augmented with one unary operator (5) or replaced by a constant or proposition (6).

Crossover is regularly combined with a mutation operator during the evolutionary search process, while the main concern is to combine LTL goals from different candidates to generate offspring. The combination function *combine* is described in Definition 17.

Definition 17 (LTL Combination). *Let ϕ and ψ be two LTL formulas. Function combine(ϕ, ψ) produces a new formula ϕ' by performing the following steps:*

1. *It selects a sub-formula α from ϕ; to be combined;*

2. *it selects sub-formula β from ψ;*

3. *it either, (a) $\phi' = \phi[\alpha \mapsto \beta]$ replaces α by β in ϕ; or (b) $\phi' = \phi[\alpha \mapsto \alpha\ o_2\ \beta]$ combines α and β with a binary operator $o_2 \in \{\vee, \wedge, \mathcal{U}, \mathcal{R}, \mathcal{W}\}$.*

To perform the operation, the genetic algorithms employ the *combine* function described in Definition 17.

For example, given $\mathcal{F}_1 : \Box(p \rightarrow q)$ and $\mathcal{F}_2 : \Diamond(r \wedge \neg p)$ as in Figure 5.3, 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 $C_1 : \Box(\neg p)$ and $C_2 : \Diamond(r \wedge (p \rightarrow q))$. This operator guarantees by construction to produce syntactically valid LTL formulas.

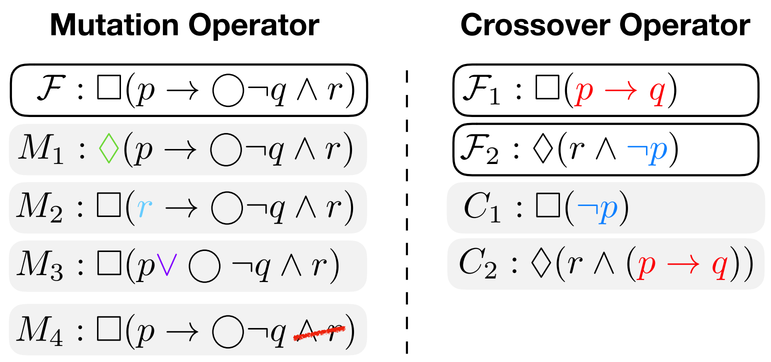

Mutation Operator
Crossover Operator
$\mathcal{F} : \Box(p \rightarrow \bigcirc\neg q \wedge r)$
$M_1 : \Diamond(p \rightarrow \bigcirc\neg q \wedge r)$
$M_2 : \Box(r \rightarrow \bigcirc\neg q \wedge r)$
$M_3 : \Box(p \vee \bigcirc\neg q \wedge r)$
$M_4 : \Box(p \rightarrow \bigcirc\neg q\ \cancel{\wedge r})$
$\mathcal{F}_1 : \Box(p \rightarrow q)$
$\mathcal{F}_2 : \Diamond(r \wedge \neg p)$
$C_1 : \Box(\neg p)$
$C_2 : \Diamond(r \wedge (p \rightarrow q))$

Figure 5.3: Mutation and Crossover operators.

5.4.5 Differential Fuzzing

SpecBCFuzz, inspired by differential testing, defines simple oracles to detect five 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 $Solver_1(\mathcal{S}' \wedge$

δ_j) N times to confirm that the solver is consistently producing the same output for the given input. If the solver always produces the same unexpected output, then the bug is confirmed. The information regarding the solver and the inconsistency pattern is added to the corresponding set. Precisely, if o_1 is sat or unsat, i.e., the solver produces an incorrect output, this is a wrong answer bug and the bug-info is added to the mentioned set. Otherwise, if o_1 is unknown or the inconsistency pattern contains an error answer because the solver always crashes with the same input, the bug-info is added to the crash or exception thrown set. The crash is a bug in which the solver stops and exits while the exception thrown explicitly shows an error message.

In the case that, after re-running multiple times, the same solver with the same input produces a different output compared to the first observed o_1, e.g., o_1 was sat, but when re-run it often produced unsat, then we consider this behavior as flaky.

For each input formula, we compute the average execution time T of the solvers to produce valid outcomes (sat or unsat), then perform 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 we will consider this as a potential bug and warn the tester by adding the corresponding bug-info to the set hang loop.

The hang loop bug is commonly found in complex inconsistency patterns. For example, tableau solvers answer sat or unsat, while a single LTL solver that is also tableau-based reaches a timeout. An additional association between different types of solvers is also considered. For example, automata-theoretic approach (ATA) and bounded model checking (BMC). Different versions of the solvers are also considered. For example, PLTL and their three different algorithms.

5.5 Empirical Evaluation

We analyze and empirically evaluate SpecBCFuzz as stated in the research questions:

- **RQ1:** What are the bugs and inconsistency patterns that SpecBCFuzz is able to find?
- **RQ2:** What are the performance issues and warning patterns that SpecBCFuzz is able to find?
- **RQ3:** How effective are seed selection and boundary condition strategies at finding inconsistencies?
- **RQ4:** Does SpecBCFuzz find more inconsistencies patterns associated with bugs than state-of-the-art fuzzing approaches?

To answer RQ1, we manually evaluated the results of differential fuzzing. First, we identify inconsistency patterns. For instance, a single difference in the outcomes, when a solver answers differently from all the others, or even all the solvers based on BMC answering differently. Second, inconsistency patterns and their failure-inducing input were used to identify bugs and take into account their behavioral observation. Interactively, all the inconsistency patterns and their association with bugs were re-evaluated and revalidated. Finally, RQ1 presents qualitative results that include bugs in LTL solvers, their types, behavioral observation, and inconsistency patterns.

Moreover, we answer RQ2 by analyzing warning patterns, in which each pattern associates a timeout warning in differential fuzzing outcomes. We observe a subset of inputs (performance-issue-inducing input) that cause problems per each type of algorithm used to implement the LTL satisfiability. The subset is potentially useful from a practical point of view to test a solver in the limits of the adopted satisfiability algorithm and to produce a dynamic approach to rebuild the input set.

Regarding RQ3, we disabled fitness functions and well-formed inputs (seed-based strategy). Thus, we compare the empirical results produced by SpecBCFuzz to check whether the fuzzing strategies themselves are able to produce close results. We evaluated the bias in the final result of SpecBCFuzz by different subsets of fuzzing strategies. Among them, we evaluated the effect of seed selection. For this purpose, we performed an unguided search in the presence of realistic specifications and their sets of boundary conditions. Moreover, we also disabled the semantic and syntax fitness functions to evaluate the effects of the boundary condition fitness under the presence of seeds previously evaluated.

RQ4 remains about a general comparison between LTLGrammarFuzz and our approach SpecBCFuzz. LTLGrammarFuzz represents a state-of-the-art approach, considering that it adopts a common fuzzing strategy that produces inputs constrained by the LTL language. Overall, LTLGrammarFuzz implements probabilistic grammar fuzzing. On the other hand, SpecBCFuzz also integrates fuzzing testing strategies as traditional seed-based inputs and also integrates boundary conditions over seeds as a new fuzzing strategy in logic domains.

In what follows, we introduce the LTL solvers with their variants, fuzzing approaches, and experimental settings.

5.5.1 LTL Solvers

The objective of this study is to evaluate the efficacy of SpecBCFuzz in identifying defects in LTL solvers that employ a variety of algorithms and heuristics with diverse representations of state and search methodologies, including automata-based or tableau. 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 behavior. In total, we use 21 solvers' versions in the evaluation, summarized in Table 5.1.

Table 5.1: 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

Aalta works on the fly and builds a satisfying LTL formula without constructing a full-state representation. Aalta version 2 [LZP+15] implements a Depth-First Search (DFS) responsible for building a temporal transition system and searching for a satisfiable trace. Specifically, the DFS creates an initial state and its successors using SAT techniques. The original formula is converted to an equivalent next normal form, and the conversion puts subformulas in terms of the operators Until and Release in the scope of the operator Next. In this way, the recursive functions are created and form the current state (propositional formula) and their successors (subformulas composed by the operator next), as mentioned. Aalta version 1 [LZP+13] also works by parsing the formula directly. The algorithm has three parts. First, the algorithm tries to identify a consistent obligation (literals that characterize a possible way of satisfying the formula) in the current state. In the case where it is identified, the formula is satisfiable. Otherwise, the algorithm tries to identify a strongly connected component in the transition system built upon the normal form extension and the recursive functions are generated by the unit and release operators. In the case it identifies, the formula is satisfiable as well. Finally, the stop condition presents the fact that the formula is the unsatisfiable case in which the algorithm explores the whole transition system.

BLACK [GGM21; GGM+21b] implements a one-pass tree-shaped tableau developed by Reynolds [Rey16a] for satisfying LTL formulas. Essentially, BLACK generates a tableau based on an LTL formula provided in the input. After that, BLACK converts the tableau for a depth k in a propositional formula based on the pruning rule. Interactively, the value of k increases until the search finds an accepted branch in the tableau or a witness of unsatisfiability is detected. The authors claim the completeness of the algorithm [GGM+21b] since it follows the pruning rule of Reynolds'tableau [GGM+21b; Rey16a]. The propositional formulas are evaluated commonly by MathSAT [CGS+13], Z3 [dMB08], CVC5 [BBB+22], or CryptoMiniSAT [SNC09].

PLTL widely integrates three algorithms in different versions. First, it implements the one-pass tableau method proposed by Schwendimann [Sch98b]. The solver builds a rooted tree or graph. After that, it checks whether the tree or graph fulfills the eventuality formulas on a branch-by-branch basis. Thus, the

LTL satisfying reduces the identification of isolated subtrees instead of strongly connected components (e.g., Aalta). The branch-based loop ensures termination. Moreover, PLTL released an implementation of Wolper's method [Wol85], and the tableau method. The algorithm starts by building a cyclic graph. After that, the algorithm conducts multiple passes while pruning inconsistent nodes and nodes that contain unfulfilled eventualities. The last released PLTL version is BDD-based. The algorithm builds subsets of the Fischer-Ladner closure [FL79] of an input formula. The algorithm then uses the greatest fixpoint approach [Mar05] to progressively delete the mentioned sets by the SAT semantic.

NuSMV implements BDD-based model checking and also explores the SAT-based model checking [CGP+02b]. NUSMV version 1 was implemented on the BDD structure. The BDD compresses formulas and subformulas obtained by the input LTL formula to deal with the state-space explosion. The BDD structure generates an implicit tableau and turns available traditional logic operators (e.g., conjunction and disjunction), where the implicit tableau enables support for the solving. NuSMV version 2 constructs an internal representation of the model based on a simplified version of the Reduced Boolean Circuit (RBC) [CCG+02]. The main notion of the bounded model checking and the RBC representation is to consider counterexamples of a particular length k and a finite prefix of a path. After that, the algorithm converts to a propositional formula, which is satisfiable iff such a counterexample exists. This constraint allows the algorithm to identify a counterexample without passing the entire search space. To improve performance, the algorithm often includes split heuristics to quickly identify the counterexample [BCC+99a].

5.5.2 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 [AMT13; DCA+18; DMR+18; DRA+16; vLDL98b; CDB+23]. They are represented in terms of domain and goal properties as indicated in the GORE methodology.

Moreover, the boundary conditions of each specification are also a seed. The boundary conditions were automatically identified during preprocessing. For this matter, we used the goal conflict identification approach in Degiovanni *et al.* [DMR+18], and once it implements a genetic algorithm, we ran the approach 10 times since random decisions can lead to a different set of boundary conditions. The final set of boundary conditions is the "weakest" boundary conditions. By definition, formula A is weaker than B, if $B \wedge \neg A$ is unsatisfiable, i.e., if B implies A. Therefore, we have guided the search process by removing all previously identified boundary conditions.

Table 5.2 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.* [DMR+18].

Table 5.2: Seeded LTL formulas and boundary conditions.

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

5.5.3 Fuzzing and Experimental Settings

SpecBCFuzz is implemented in Java with the JMetal framework [NDV15] that allows instantiate different optimization algorithms. NSGA-III runs the search process as presented in Section **??** and the mutation and crossover operators presented. For selection, a four-solution tournament was integrated. The population size of 100 individuals was defined, and the fitness evaluation was limited to a number of 1000 individuals. The probability of crossover application was 0.1, while mutation operators were always applied. Moreover, the timeout of the model counting as 30 seconds. The model counting also received a bound of 40. Regarding the consistency of the LTL grammar, SpecBCFuzz uses the OwL library [KMS18] to parse and manipulate the LTL specifications, restricted for future LTL operators. We make our tool, seeded formulas, and boundary conditions publicly available at: **`https://github.com/lzmths/SpecBCFuzz`**.

The baseline approach is a probabilistic grammar fuzzing that also uses the OwL library and its LTL grammar constrained for future LTL operators. The parameters are the maximum of literals per each formula (ranging from 1 to 9), the maximum number of non-terminal reached (ranging from 2 to 10), terminal choice probability (from 0.20 to 0.44), boolean constant probability (ranging from 0.05 to 0.29).

We ran SpecBCFuzz and probabilistic grammar fuzzing experiments 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 Redhat Linux, version 7. Moreover, we also set two days as the time threshold for each execution of each job. Each job has a time limit of 48 hours. In total, we launched 25 jobs in the mentioned HPC cluster. In the case of SpecBCFuzz, we provide a different specification and its associated boundary conditions set per each job, while the probabilistic grammar fuzzing varied its parameters between maximum and minimum values with a continuous increment of 0.01 for probability and 1 for absolute numbers. Furthermore, we conducted the experiment process 10 times to reduce random elections of the search algorithm and probabilistic grammar fuzzing. In addition, the LTL solvers were configured with a timeout of 300 seconds.

We also conduct statistical analysis, namely, the Wilcoxon signed rank test ($p - value$) [Wil45] and the Vargha-Delaney measure $\hat{A}_{12}$ [VD00], to compare the different fuzzing approaches. They evaluate SpecBCFuzz in relation of variants and the baseline. Moreover, we also compute the binary vector dissimilarity [ZS03]. In this way, we compute the Sørensen–Dice coefficient [SOR48; Dic45] to compare the similarity measure of warning patterns and satisfiability search algorithms.

5.6 Evaluation

5.6.1 Bugs and Inconsistency Patterns

Table 5.3 introduces bugs found in our differential fuzzing evaluation. Each row in Table 5.3 is a bug with additional information such as its type (e.g., flaky and hang loop), inconsistency pattern associated with the bug case, and the affected LTL solver. Furthermore, Table 5.4 shows the inconsistency patterns associated with bugs and identified in the experimental results (solvers, input formulas, and answers of solvers). Moreover, Table 5.4 also presents the ratio of whether each inconsistency pattern case occurs at least once per approach execution.

For example, the first row presents a parser failure in the Black solver, version 0.5.2. The type of bug is cataloged as an exception thrown since the solver shows the outcome: *"syntax error: <stdin>: Expected ')' "*. Table 5.4 shows the details of inconsistency patterns. In this case, the pattern is the inconsistency of black solvers in version 0.5.2 associated with the operator weak-until and parenthesis in the beginning and in the end. A reduced input is (aWb).

In general, the empirical evaluation identified 16 bugs in the evaluated LTL solvers, where five types of bugs are coded in the qualitative analyses of the inconsistency patterns. The types of bugs are (i) exception thrown, (ii) crash, (iii) flaky, (iv) hang loop, and (v) wrong answer.

The exception thrown is an abnormal error message display. For instance, a syntax error occurs when the input formula follows the grammar defined by the solvers as aforementioned in the example. The crash is a bug once the LTL solver stops and exits. The crash and exception throw are commonly associated, as in

Table 5.3: Bugs found in the LTL solvers

Bug ID	Bug Type	Bug Description	IP ID	Solvers	Solver Versions
1	Exception thrown	Syntax parser failure	1	Black	0.5.2
2	Wrong answer	Operator G and the negation of its operand always produces wrong answers.	2	PLTL	BDD
3	Crash and Exception thrown	Crash and throw the message: *Segmentation fault (core dumped)*	3	NuSMV (BMC)	2.5.4 and 2.4.3
4	Crash and Exception thrown	Crash and throw the message: *Segmentation fault (core dumped)*	4	NuSMV (BMC)	2.6.0, 2.5.4, and 2.4.3
5	Crash and Flaky	Crash without error message	5	NuSMV (BMC)	2.4.3
6	Crash and Exception thrown	Crash and throw the message only in version 1: *Assertion failed.*	6	Aalta	2 and 1
7	Flaky and Wrong answer	Altaa presents a wrong answer time to time	7a,7b	Aalta	2 and 1
8	Exception thrown	Black runs and throws the message: *Killed*	8	Black	All
9	Hang loop and Exception thrown	Version 2 does not answer in 24 hours and version 1 throws and assertion failed (*qi check*)	9	Aalta	2 and 1
10	Hang loop	The solver does answer in 24 hours	10	PLTL	Tableau
11	Exception thrown	The solver throws the message: *Segmentation fault (core dumped)*	11	PLTL	BDD
12	Flaky and Crash	The solver throws the message time to time: *Segmentation fault (core dumped)*	12	NuSMV (BMC)	2.6.0 and 2.5.4
13	Flaky, Crash, and Hang loop	Version 2 crashes without answer and Version 1 throws Assertion failed	13	Aalta	2 and 1
14	Crash and Hang loop	Version 2 does not answer in 24 hours and Version 1 throws assertion failed (*clsnum check*)	14	Aalta	2 and 1
15	Hang loop	Black does not answer in 24 hours	15	Black	All
16	Wrong answer	Aalta's answers differ from all others	16	Aalta	2 and 1

bug 3.

A flaky bug occurs when the solvers do not produce the same result every time. The analysis of inconsistency patterns in the empirical evaluation points to this bug in a single inconsistency of an LTL solver. For example, Bug 7 is associated with inconsistency patterns 7a and 7b insofar as Aalta produced a single inconsistency in version 2 for a set of failure-inducing inputs, while version 1 is associated with another set that contains a single inconsistency.

The validation of the pattern confirmed that both patterns are associated with the same bug 7 since the reexecution produces the wrong answer of the Aalta solvers. The flaky bug is confirmed by the wide reexecution. Among the failure-inducing input identified by the inconsistency patterns 7a and 7b, Aalta produces, for instance, the wrong answer in 1 out of 1000 executions. In this case, the wrong answer is confirmed by the consensus of the other solvers.

A hang loop bug is observed when running without stops until an extended time threshold. Thus, the timeout warnings were manually analyzed to identify potential hang loops. In general, the results in some cases indicate a potential performance warning. For instance, the solver answered an LTL formula after one hour, and the other solvers answered in less than 1 second. However, inconsistency patterns associated with bugs produce timeout warnings after a long extension. That is, the buggy solver does not produce an answer after 24 hours, and other solvers have still answered in a few seconds. Additionally, the formula contains less than one hundred clauses. In these terms, the inconsistency pattern 10 associated with bug 10 indicates a bug because the PLTL (version tableau) presents timeout and black solver failures (version 0.5.2) in the presence of parentheses and the *weak-until* operator. Inconsistency pattern 10 also associates the pattern match rule of inconsistency pattern 1 since parenthesis and the operator *weak until* are related to the hang loop source.

All four mentioned types of bugs are identified by the inconsistency pattern and manually validated by behavioral observation, such as memory usage and behavioral presentation with an extended and wide time threshold (24 hours), while the fifth type of bug *wrong answer* happens insofar as there is no consensus in the LTL solvers. Moreover, the confirmation is based on satisfiability preserving transformation when applied to the formula. Transformations are based on the generalization inference rule [GRT18].

Table 5.4: Inconsistency patterns associated with bugs

IP ID	Description	Occurrences	
		ACoRe	LTLGrammarFuzz
1	Black solver (version 0.5.2) output presents inconsistency when the LTL formula contains the operator *weak until* and double parenthesis at the beginning and also in the end.	100%	100%
2	PLTL (version bdd) output presents inconsistency when compared to the other solvers.	70%	100%
3	NUSMV (bmc and version 2.5.4) runs and throws an error.	100%	100%
4	NUSMV (bmc, versions 2.5.4 and 2.4.3) runs and throw an error.	100%	100%
5	NUSMV (bmc, version 2.4.3) runs and throws an error.	100%	50%
6	Aalta runs and throws an error.	100%	70%
7a	Aalta (version 1) output presents inconsistency when compared to the other solvers.	100%	100%
7b	Aalta (version 2) output presents inconsistency when compared to the other solvers.	100%	100%
8	NuSMV (bmc and version 2.6.0) has a timeout and previous versions throw errors. Also, Aalta (version 1) is inconsistency and all versions of Black solver reach a timeout.	100%	0%
9	Aalta (version 2) output presents inconsistency when compared to the other solvers. and Aalta (version 1) throws an error.	100%	0%
10	Pattern 1 and PLTL solvers (versions tableau and multi-pass) reach timeouts.	30%	100%
11	PLTL (version bdd) throws an error.	0%	100%
12	Aalta (version 1) output presents inconsistency when compared to the other solvers and NUSMV (bmc, version 2.5.4) runs and throws an error	100%	40%
13	Aalta (version 2) throws an error.	80%	0%
14	Aalta (version 2 and version 1) reach timeouts.	100%	100%
15	NUSMV (bmc and version 2.6.0 and 2.4.3) reach two timeouts, NUSMV (bmc and version 2.5.4) throws an error, and all versions of Black solvers reach timeouts.	100%	100%
16	Pattern 1 and Aalta (version 1 and 2) presents inconsistency when compared to the other solvers.	100%	30%

In general, exception thrown tracks 44% of bugs and crashes also presents 44%. Moreover, the hang loop presents 31% of the bug, while flaky is responsible for 25%. Finally, the single inconsistency instantiated by the wrong answer catalog is attributed to 12% of the bugs. SpecBCFuzz is able to find 15 out of 16 bugs, while LTLGrammarFuzz is able to identify 12 out of 16 bugs in 10 executions. In what follows, we compare the effect of seed approaches in these results, the advantages of the boundary conditions, and we compare the occurrence distribution of inconsistency patterns.

Moreover, we confirm the bugs found by SpecBCFuzz. First, we conduct a double check for all bugs. This means that we are able to reproduce the bugs in a similar environment. Moreover, we also confirmed the bugs with the solver

Table 5.5: Bugs fixed, confirmed by theoretical properties, or confirmed by developers.

Bug ID	**Solvers**	**Solver Versions**	**Fixed**	**Theoretical**	**Bug report**
1	Black	0.5.2	✓		
2	PLTL	BDD		✓	
3	NuSMV (BMC)	2.5.4 and 2.4.3	✓		
4	NuSMV (BMC)	2.6.0, 2.5.4, and 2.4.3			
5	NuSMV (BMC)	2.4.3	✓		
6	Aalta	2 and 1			
7	Aalta	2 and 1			✓
8	Black	All			✓
9	Aalta	2 and 1			✓
10	PLTL	Tableau		✓	
11	PLTL	BDD		✓	
12	NuSMV (BMC)	2.6.0 and 2.5.4			
13	Aalta	2 and 1			✓
14	Aalta	2 and 1			✓
15	Black	All			✓
16	Aalta	2 and 1			✓

developers, thereby bug reports, theoretical properties such as generalization of LTL and tableau build, or even fixed bugs in the previous version.

We submitted the bug for active projects of LTL solvers, and we have received an answer for the Black and Aalta projects. The other bugs were checked by comparison based on the existence of the bug in the new releases. After that, we checked the bugs by theoretical properties that should be maintained by the solver. For example, including theoretical generalization properties such as adding a global operator at the beginning of the formula should preserve equisatifiability [GRT18]. That is, if the original formula is satisfiable, then the generalized formula should confirm the satisfiable status. In the case of a bug such as the wrong answer, the buggy solver is not able to preserve equisatisfiability.

Moreover, we adopted theoretical metamorphic properties based on tableau. For example, if PLTL builds a tableau and answers for a given LTL formula, then Black should also answer for the same formula since it also builds a tableau and completes the evaluation with a BMC heuristic for LTL satisfiability. Table 5.5 presents the results of the bug confirmation by type of confirmation. In total, we are able to double check (i.e., reproduce the bugs in a similar environment) for all bugs. External confirmations were conducted positively to 13 out of 16 bugs, while internal confirmation by re-executing bugs can confirm all the cases.

Quantitative comparisons of their distributions are presented in the following evaluations.

5.6.2 Performance Issues and Warning Patterns

Warning pattern associates timeout warnings in a set of one or more solvers. They are identified by a single timeout warning, a combination of them, or even all the solvers, resulting in a timeout warning. Performance warning patterns are particularly useful for testing purposes since they show performance issues in a particular LTL solver, version, or even implemented algorithm. Thus, the set of warning patterns has the potential to indicate improvements in a particular solver for releases, heuristics set, or theoretical guidelines for a particular type of satisfiability algorithm.

In total, the empirical evaluation reveals that 315 warning patterns were produced by SpecBCFuzz, while LTLGrammarFuzz identifies 207 patterns. Thus, we analyzed the associations that came from SpecBCFuzz experiments. Among them, we seek potential associations, such as LTL solvers with the same algorithm or the presence of performance issues in the different versions. In the case of algorithms, we separated the algorithms into three types: (i) BDD, (ii) BMC, (iii) tableau. BDD is present in NuSMV (integrated component in version 1) and PLTL (version BDD). Tableau algorithms are the basis of the Black and PLTL (graph-based and multipass-based), while the automata-theoretic approach is implemented by Aalta (version 1). Moreover, BMC notions are found in NuSMV (an integrated component in version 2) and Aalta (version 2).

We evaluate the binary association of timeout warnings with regard to the latest releases of the LTL solvers. Specifically, we compare when solvers reach the timeout warning together in the same inputs, and each one reaches the timeout warning in different inputs. For this purpose, we compute the Sørensen–Dice coefficient [SOR48; Dic45] that is a similarity measure for binary association.

Sørensen–Dice coefficient observes in a pair the following conditions: (i) true positive, (ii) false positive, and (iii) false negative [SOR48; Dic45]. In our analysis, the true positive represents the cases where two solvers reached the timeout warning in the same input, while false positives represent the case where a single solver reached the timeout. The false negatives are the cases where both solvers properly solve the satisfiability problem.

Table 5.6: Sørensen–Dice coefficient of warning patterns

LTL Solvers	**NUSMVbmc**	**NUSMVbdd**	**Aalta (version 2)**	**PLTL graph-based**	**PLTL multipass-based**	**PLTL bdd**	**Black**
NUSMVbmc	0.5	0.0	0.035	0.00786	0.00199	0.0	0.18
NUSMVbdd	0.0	0.5	0.0	0.0	0.0	0.0	0.0
Aalta (version 2)	0.035	0.0	0.5	0.101	0.0190	0.0	0.115
PLTL graph-based	0.00786	0.0	0.101	0.5	0.176	0.0	0.034
PLTL multipass-based	0.00199	0.0	0.0190	0.176	0.5	0.0	0.0087
PLTL bdd	0.0	0.0	0.0	0.0	0.0	0.5	0.0
Black	0.18	0.0	0.115	0.034	0.0087	0.0	0.5

First, the strongest association is between the BDD solvers. The BDD solvers do not provide a single timeout during execution. Second, the Sørensen–Dice's coefficient from 0.18 to 0.10 suggests the similarity: (Aalta;PLTL graph-based), (PLTL graph-based;PLTL multipass-based), (NuSMVbmc;Black), and (Black;Aalta). Third, the Sørensen–Dice's coefficient also points out the similarity from 0.02 to 0.04:

(PLTL graph-based;Black), (Aalta;PLTL multipass-based), and (NuSMVbmc;Aalta). Table 5.6 presents additional relations. Concerning the three association levels mentioned, we observe three groups of solvers. BDD-based is composed of NuSMV and PLTL (BDDs-based). The tableau group is made up of Aalta (version 2), PLTL (graph-based), and PLTL (multipass-based). Finally, the third group is the BMC, composed of NuSMV (bmc), Black, and Aalta (version 2).

As expected, the three main groups of algorithms were identified to implement a solver. Furthermore, the results suggest two solvers at the intersection of BMC and tableau: Black and Aalta. In fact, Aalta (version 1) implements an automata-theoretic algorithm [LZP$^+$13], which the similarities of a tableau is known. Among them, a tableau may define finite automata on infinite trees, and an opposite relation is also possible [Eme85]. The relation of Aalta (version 2) with the BMC algorithm is observed in the successor notion of the depth-first search as claimed by their authors [LZP$^+$15]. In the case of Black, the tableau algorithm is also implemented. However, many pruning rules are adopted as heuristics and are based on BMC notions. For example, Black encodes in a boolean formula up to depth k, increasing k in a loop. For this reason, Black was empirically associated as a BMC-like solver.

From a practical point of view, the potential developers and testers of the solvers may produce two kinds of performance-issues-inducing input. First, the input produces a performance issue in a particular solver. The second kind of inputs are inputs relevant to satisfiability algorithms, and they were detected by the Sørensen–Dice's coefficient.

5.6.3 Seeds and Boundary Conditions Matters

For the evaluation of the influence of fuzzing strategies in SpecBCFuzz, we disabled fitness functions and well-formed inputs. First, we intend to compare the effect of seed selection, and then we conduct an unguided search on top of the LTL specifications and their boundary conditions (seed-based strategy). Second, we also disabled the semantic and syntax fitness functions and evaluated the ability of the boundary condition reduction fitness to guide the search process to find inconsistency patterns.

The empirical experiment points out inconsistency pattern 4 (IP4), while the seed strategy is instantiated. The search guided by boundary conditions also finds IP4, and it additionally identifies IP1. The statistical evaluation of the occurrence of IP1 and IP4 shows that the guided search by the boundary condition notion is significantly larger than the unguided search with seeds in the inputs. The p-value is lower than 0.00001, and the size effect is equal to 1.0.

Furthermore, we compute the occurrence of warning patterns identified by SpecBCFuzz and the impact caused when fuzzing strategies are disabled. Analysis of warning patterns also indicates the advantage of SpecBCFuzz. Observation statistical evaluation also computes lower p-values than 0.00001.

We conclude the importance of adopting traditional fuzzing strategies such as seeds and the fitness of reduction of boundary conditions (guided search with fitness is able to produce complex inputs). However, the empirical evaluation presents the view that the strategies should be combined. In our study, maintaining the constrained semantics and syntax of representative inputs is also important, as shown in the RQ1 results. Especially, the combination of the constrained semantics and syntax with the traditional fuzzing strategies as seeds and the evolutionary process.

5.6.4 Statistical Analysis of SpecBCFuzz

Table 5.7: Patterns of inconsistency and their occurrence described by minimum, maximum, average, and effect size ($\hat{A}_{12}$)

IP ID	**ACoRe**				**LTLGrammarFuzz**			
	min	max	avg	$\hat{A}_{12}$	min	max	avg	$\hat{A}_{12}$
1	8023	9385	8552	0.0	30226	30673	30383.8	**1.0**
2	0	27	7.5	0.0	7433	7689	7570.7	**1.0**
3	455	586	529.5	**1.0**	14	32	19	0.0
4	4288	4556	4404.3	**1.0**	58	80	74	0.0
5	31	66	49	**1.0**	0	4	0.9	0.0
6	49	88	67.8	**1.0**	0	5	1.3	0.0
7a	454	494	468.8	**1.0**	3	13	8.5	0.0
7b	411	517	459.6	**1.0**	5	12	8.7	0.0
8	5	17	8.6	**1.0**	0	0	0.0	0.0
9	4	15	9.4	**1.0**	0	0	0.0	0.0
10	0	1	0.3	0.0	5	24	14.5	**1.0**
11	0	0	0.0	0.0	44	70	55.5	**1.0**
12	26	43	35.1	**1.0**	0	3	0.6	0.0
13	0	2	1.0	**0.9**	0	0	0.0	0.1
14	238	307	278.8	**1.0**	2	9	4.3	0.0
15	1	12	7.2	**0.795**	2	5	3.4	0.205
16	4	16	7.8	**1.0**	0	1	0.3	0.0

For each inconsistency pattern, we compute the effect size between the occurrences found by SpecBCFuzz and LTLGrammarFuzz. The effect size allows us to evaluate the strength of each approach per inconsistency pattern. Moreover, the general capability to find patterns and failure-inducing inputs. For this measurement, we compute the non-parametric effect size Vargha and Delaney's $\hat{A}_{12}$ [VD00].

From Table 5.7 we can observe that SpecBCFuzz outperforms LTLGrammarFuzz. In 17 inconsistency patterns, SpecBCFuzz finds the strong effect size measurement in 13 out of these patterns. Moreover, Table 5.7 also presents the minimal, maximal, and average occurrence of each inconsistency pattern (quantitative cases of failure-inducing inputs). In general, we also can observe that the size effect of all inconsistency patterns is 0.70, which means that SpecBCFuzz is stronger in the general task of finding the occurrence of inconsistencies than LTLGrammarFuzz.

Regarding significance testing, we compare the occurrence of inconsistency patterns in terms of the non-parametric statistical test Mann-Whitney U test [MW47]. We set the *alpha*-value as 0.05, and the statistical test realizes a p-value lower than 0.00001. Thus, the test evidence that SpecBCFuzz leads to significant occurrences of an inconsistency pattern compared to LTLGrammarFuzz.

As mentioned, SpecBCFuzz identifies more warning patterns than LTLGrammarFuzz. To quantitatively compare both approaches in terms of the occurrence of warning patterns, we computed the significance testing of the occurrence in terms of the same statistical test conducted in the evaluation of inconsistency patterns. In this way, the comparison presents a p-value lower than 0.00001. Thus, the one-sided alternative hypothesis suggests that the occurrence distribution identified by SpecBCFuzz is the largest.

5.7 Threats to Validity

A threat related to our study is the adopted third-party tools. To mitigate potential problems related to that, SpecBCFuzz relies upon reliable frameworks and libraries. Regarding the optimization steps, SpecBCFuzz integrates the JMetal framework [NDV15] and calls NSGA-III [DJ14]. The inputs are parsed and handled by the OWL library [KMS18]. All the libraries and frameworks mentioned have been adopted in different studies, and the results are also consistent.

Moreover, the implementation steps concern components such as fitness evaluation and differential comparison. Bugs contained in these components might affect the empirical results. For instance, they can cause false-positive evidence for a potential inconsistency pattern or even a bug. To mitigate these threats to validity, we manually double-checked the differential warnings and the derived results, such as inconsistency patterns and bugs. A double check was performed for each level of empirical refinement.

The bugs were double-checked depending on their types. The wrong answers were confirmed by the transformation that preserves equisatisfiability. Specifically, we applied the mentioned transformation until we confirmed that the original answer changed, despite the equisatisfiability preserving. The crash and exceptions were checked by simple reproduction, while the flaky response was confirmed by extensive reproduction and the wrong answer was observed in a subset of execution. We confirmed the hang loop type by the reproduction and the timeout extension of 24 hours.

External validity is reduced because we experimented with different solvers, versions, data structures, and LTL satisfiability core algorithms. In addition, the inputs or seeds of SpecBCFuzz were widely extracted by different specifications and sources, while the boundary condition was extracted from different executions. In general, we mitigate the reproducible execution of the experiment by running tools or components several times with random features.

5.8 Conclusion

LTL solvers are built using a variety of sophisticated satisfiability algorithms. The current evaluation of algorithms presents a performance-oriented evaluation. However, software testing techniques that identify bugs are typically absent, whereas there are testing approaches for SAT and SMT solvers. For this reason, we propose SpecBCFuzz as a search-based fuzzing approach that is effective in identifying relevant failure-inducing inputs. This is achieved when SpecBCFuzz is optimized and searched using boundary conditions, syntactic and semantic similarity, and general satisfiability of the modified LTL specifications. Furthermore, our analysis demonstrates that SpecBCFuzz outperforms a probabilistic grammar fuzzer that was specifically designed for fuzzing LTL solvers.

In addition, a comparative analysis was conducted to evaluate the empirical results generated by SpecBCFuzz. This analysis aimed to determine whether the fuzzing approach itself is capable of producing similar outcomes when the fitness functions and seeds are disabled. In this manner, the bias in the final result of the tool was evaluated using different subsets of fuzzing strategies. To this end, an unguided search was conducted in the presence of realistic specifications. Furthermore, the semantic and syntax fitness functions were disabled to assess the impact of boundary condition fitness in the presence of previously evaluated seeds. The empirical results suggest that reducing the fitness function also reduces the capability of SpecBCFuzz.

In the course of this search process, we claim that SpecBCFuzz is capable of generating failure-inducing inputs, given that boundary condition verification transitions from logical inconsistency in the unsatisfiable problem to a satisfiable answer. In other words, we utilize LTL solvers to their limits of satisfiability. Furthermore, additional fitness criteria, such as semantic and syntactic similarities, can constrain the search space and maintain the seeds provided by the specifications as representative of realistic LTL formulas. In addition, the algorithm determines the satisfiability of the modified specification. In general, the empirical evaluation identified 16 bugs in the LTL solvers evaluated, where five types of bugs are coded in qualitative analyses of inconsistency patterns.

Moreover, our evaluation encompasses the examination of warning patterns, which represent a guide for associating timeout warnings. Warning patterns are particularly useful for performance testing purposes, as they demonstrate performance issues in a specific LTL solver, version, or even an implemented algorithm. Consequently, the set of warning patterns has the potential to indicate improvements in all the mentioned levels. Our empirical result shows that SpecBCFuzz is able to identify relevant warning patterns for performance testing.

6

Fuzzing Linear Temporal Logic Portfolio

Linear Temporal Logic (LTL) solvers have been proposed to address the LTL satisfiability problem. In an effort to answer the satisfiability problem, the LTL solvers were designed around several algorithms. Among the additional efforts made, developments have also been made in terms of improving performance gains. Among the applications are complex contexts such as program analysis, software verification, symbolic execution, program synthesis, model checking, and artificial intelligence reasoning. In such contexts, it is uncommon to identify the best solution in terms of performance. A particular satisfiability algorithm may offer the best performance in a particular problem, while it has a poor performance for another. A potential solution for the LTL satisfiability problem is the algorithm portfolio. That is, the strategy involves the execution of algorithms in parallel with the objective of achieving better distribution performance. In summary, the algorithms portfolio exploits the lack of correlation in the performance of multiple algorithms to achieve better performance. Overall, it achieves better performance in the average case. However, implementations of satisfiability problems have been reported as tools that contain bugs. To address this problem, we modify portfolio decisions to incorporate a confirmation answer. In other words, the portfolio does not provide the fastest answer. The portfolio replicates the answer when at least two solvers agree on a given input. It was formulated based on a previous study that identified bugs in LTL solvers and demonstrated that a considerable number of these bugs are not inherent to a single input across different solvers. Our empirical results present bugs in the pure portfolio, whereas a search-based fuzzing approach does not find a single bug for the reliable portfolio. Furthermore, we also evaluated the reliable portfolio in terms of performance. The empirical results present the lowest runtime for the reliable portfolio compared to a single robust LTL solver (NuSMV). In summary, a reliable portfolio is capable of increasing soundness in LTL solvers and is also associated with a gain in performance compared to an individual LTL solver that contains a high level of soundness.

Contents

6.1 Introduction 107

6.2 Background 109
6.3 Portfolio 112
6.4 Fuzzing 116
6.5 Families of LTL Benchmarks 117
6.6 Empirical Study 118
6.7 Results 121
6.8 Threats to Validity 124
6.9 Conclusion 125

6.1 Introduction

Linear temporal logic (LTL) is a powerful logical formalism that includes descriptive operators for future events that deal with concurrent behaviors. Moreover, LTL also enables support for verifiability, since LTL provides a formalism for the description of correctness properties for state transition systems [CHV$^+$18; Pnu77]. LTL has a general set of operators that impose a high level of expressiveness and complexity. The formalism includes safety, liveness, and fairness properties [OL82]. Additionally, expressiveness may be bounded by fragments such as GR(1) [PPS06] or well-known specification patterns [DAC99].

LTL is commonly used as a specification language for reactive systems. That is, systems that have to react to an environment that cannot wait [HP85], such as databases, control systems, and interactive systems. Moreover, model-checking tools frequently use this formalism for the formal verification of safety-critical systems such as hardware design, communications systems, and financial systems [Roz11]. Among them, we may state TLA+, Spin, and NuSMV. That contains LTL and additional specification or model languages for distributed, concurrency, and hardware systems [Lam02; CCG$^+$02; Hol97].

Furthermore, LTL is also a formalism to describe a requirement by properties that the system-to-be should maintain or reach [vLam09]. A common way to describe formal requirements is based on Goal-Oriented Requirements Engineering (GORE). In this abstraction, requirements have domain properties, goals, and stakeholders. The domain properties usually contain natural laws that the systems-to-be should respect. At the same time, stakeholders propose goals as targets that the systems-to-be should reach, in some sense, from a selfish perspective. That is, without observing the whole system or conflicts between them. In this manner, the requirement conflict analysis also employs the use of LTL. Thus, LTL is a formalism for identify and repair conflicts among the mentioned requirement properties [DvLF93].

For the mentioned reasons and applications, LTL solvers have been built over time. In general, LTL solvers compete for time performance and application specificity. Thus, it is uncommon to find the best algorithm or heuristic for the LTL satisfiability problem. In that way, the portfolio may present a general solution since the portfolio intends to associate different LTL solvers and run them in parallel. For the portfolio notion, the solvers can be combined and consequently build a resulting portfolio with a low average runtime.

Portfolio design decisions are widely debated in the community [XHH$^+$12]. Among them, the diversity principle states algorithm diversification that employs different algorithms, data structures, searches, or implementations. It aims to reach different behaviors. Thus, each portfolio member exploits the best time performance for a particular subset of a given problem. However, the portfolio imposes a potential risk, since combining LTL solvers in a single portfolio can also combine previous bugs contained in each one. Previous studies report the presence of bugs in LTL solvers by static and dynamic approaches [SD11; CDC$^+$24]. Moreover, SAT and SMT have also been reported in the literature as solution tools that contain bugs [BLB10; BMB$^+$18; SYW$^+$23].

Therefore, we modify portfolio decisions to introduce a confirmation answer. That is, the portfolio does not answer for the fastest answer. However, it answers in the first confirmation answer. That is, the portfolio replicates the answer when at least two solvers provide an agreement for a given input. Moreover, the new portfolio decision allows us to increase the level of confirmation answers. The new set of portfolio decisions is called *reliable portfolio*. It was based on a previous study [CDC$^+$24] that presents bugs in LTL solvers and shows that many bugs are not common for a single input in different solvers. Initially, the confirmation of two answers should be enough to considerably reduce the presence of bugs in LTL solvers and the resulting portfolio.

The reliable portfolio is evaluated according to SpecBCFuzz, which represents the state of the art fuzzing approach for the identification of bugs in LTL solvers [CDC$^+$24]. In general, SpecBCFuzz is a search-based fuzzing technique. Consequently, SpecBCFuzz generates LTL formulas through a search process, with the resulting offspring evaluated according to a fitness function. SpecBCFuzz accepts as input a set of seeds constructed according to formal requirements expressed in LTL and a set of divergence cases for the mentioned formal requirements, designated as boundary conditions. The objective of the search process is to evolve the LTL specification into repairs that remove boundary conditions.

During the search process, SpecBCFuzz evaluates four fitness values, which produce complex inputs for LTL solvers and also constrain the search space. These values are as follows: the satisfiability of the LTL repair, the logical inconsistency of boundary conditions, semantic similarity, and syntax similarity. Subsequently, SpecBCFuzz generates complex inputs for LTL solvers and builds differential fuzzing outcomes. This is due to the fact that different LTL solvers are called sequentially by SpecBCFuzz. The differential fuzzing result is then manually analyzed to investigate the presence of bugs by outliers cases.

Furthermore, we also evaluate the reliable portfolio in terms of runtime performance. We claim that introducing an answer confirmation reduces runtime performance. However, the average runtime for a reliable portfolio is lower than for a single and robust LTL solver. For that purpose, we empirically validate the performance of a reliable portfolio and a single robust LTL solver. Thus, we select the Schuppan and Darmawan families of LTL benchmarks [SD11]. The benchmarks evaluate LTL satisfiability in terms of runtime performance. It also includes previous benchmarks for a wide range of origins, communities, and specifications.

In general, the families of benchmarks guide the performance evaluation for many different cases and the space complexity of the LTL formulas.

Our empirical evaluation reports that SpecBCFuzz find bugs in the LTL portfolio. Moreover, the empirical evaluation presents that portfolio combines soundness and flaky bugs. That is, wrong answers given by a solver and different responses across re-runs of the solver on the same formula. In addition, previously reported bugs, such as exception throws, crashes, or performance bugs, are mitigated by the portfolio. Moreover, SpecBCFuzz is unable to find a single bug in the reliable portfolio. This is motivated by the fact that the bugs occur for different inputs and solvers.

Regarding runtime performance, our empirical evaluation presents a reliable portfolio with runtime performance lower than NuSMV (bdd version) that represents a single robust LTL solver, since previous evaluations are unable to show bugs [CDC+24]. Additionally, the occurrence of timeouts is significantly reduced for a reliable portfolio.

In what follows, Section 6.2 presents the background. Section 6.3 introduces the portfolio and algorithm selection problem, while Section 6.4 presents SpecBCFuzz and Section 6.5 introduces Schuppan and Darmawan families of LTL benchmarks. The empirical study is reported in Section 6.6 and the empirical results in Section 6.6. Moreover, the related work is discussed in Section 2.4, and threats to validity are discussed in Section 6.8. Finnaly, Section 6.9 presents the conclusion.

6.2 Background

6.2.1 Linear Temporal Logic

LTL formulas are inductively defined using the standard logical connectives, and the temporal operators to describe future events. For instance, the boolean connective ($\vee$) and the traditional definitions for *true* and *false*. For future events, we commonly define next ($\bigcirc$) and until ($\mathcal{U}$). For a formula φ and position $i \geq 0$, we say that φ holds at position i of σ. Thus, we write $\sigma, i \models \varphi$. Thus, LTL formulas are defined as follows [CHV+18]:

Definition 18 (LTL Syntax)**.** *Let AP be a set of propositional variables:*

1. *constants true and false are LTL formulas;*
2. *every $p \in AP$ is a LTL formula;*
3. *$p \in AP$ and $\sigma, i \models p$ iff $p \in \sigma_i$;*
4. *$\sigma, i \models \neg\varphi$ iff $\sigma, i \not\models \varphi$;*
5. *$\sigma, i \models \varphi \vee \phi$ iff $\sigma, i \models \varphi$ or $\sigma, i \models \phi$;*

6. *$\sigma, i \models \bigcirc\varphi$ iff $\sigma, i+1 \models \varphi$;*

7. *$\sigma, i \models \varphi\mathcal{U}\phi$ iff there exists $n > i$ such that $\sigma, n \models \phi$ and $\sigma, m \models \varphi$ for all m, $i \leq m < n$;*

8. *if φ and ϕ are LTL formulas, then are also LTL formulas $\neg\varphi$, $\varphi \vee \phi$, $\bigcirc\varphi$, and $\varphi\mathcal{U}\phi$.*

We also consider other typical connectives and operators, such as, $\wedge$, $\Box$ (always), $\Diamond$ (eventually) and $\mathcal{W}$ (weak-until), that are defined in terms of the basic ones. That is, $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$, $\Diamond\phi \equiv \mathit{true}\,\mathcal{U}\ \phi$, $\Box\phi \equiv \neg\Diamond\neg\phi$, and $\phi\mathcal{W}\psi \equiv (\Box\phi) \vee (\phi\mathcal{U}\psi)$.

6.2.2 Satisfiability

The notion behind an LTL model checking is satisfiability calls for finding a model that satisfies a logical formula [CHV+18], while the LTL solver answers if a formula contains at least one trace or model that satisfies the formula, namely SAT answer. Otherwise, the LTL solver should answer UNSAT. Thus, we define LTL satisfiability:

Definition 19 (Satisfiability)**.** *An LTL formula φ is said* satisfiable *(SAT) iff there exists at least one trace satisfying φ.*

Regarding the computational complexity of satisfiability, it is established that in the case of LTL with the operators, eventually, always, next, until, and since, satisfiability is a PSPACE-complete [SC85]. The complexity of satisfiability is NP-complete when the set of temporal operators is restricted [SC85]. Previous studies show that all relevant cases are either PSPACE-complete or NP-complete [BSS+07]. For instance, non-trivial tractability, as well as the smallest possible sets of propositional and temporal operators that have been shown to result in NP-complete or PSPACE-complete [BSS+07].

LTL solvers have been proposed to address the satisfiability problem stated in Definition 19. In an effort to answer the satisfiability problem, the LTL solvers were designed around several algorithms and data structures. Among the additional effort ends, the developments also address performance improvement [SD11], diversity of temporal operators [GGM+21a], and expressiveness [Lam02].

6.2.3 Fuzzing

Fuzzing, or Fuzz testing, is a powerful technique to generate random input strings [MFS90]. Moreover, fuzzing can perform different types of tests [ZWC+22] such as system and integration tests, open-box tests, or even semantic-oriented tests [PLS+19; SZ22]. Bugs may be found depending on the parameters provided and engines instantiated. Furthermore, fuzzing addresses target software that may concern non-functional requirements such as security [FME+20; BMA+22].

Traditional Fuzzing is able to find bugs in syntax parse modules. However, they hardly ever find bugs in deeper functionality [PLS$^+$19]. For instance, the bugs found in semantic modules as non-conformance to specifications. In this order, different engines and strategies have been proposed.

Among the successful fuzzing engines, American Fuzzy Lop (AFL) has become one of them for automated security bugs. Moreover, the AFL influenced many variants of engines and also strategies. For instance, AFLFast extends in grey-box fuzzing and represents coverage of the system under testing as a Markov chain [BPR19], AFLGo defines an inter-procedural measure of distance and search by an annealing simulation [BPN$^+$17], and Zest implements a property-based notion [PLS$^+$19; PLS19]. Among the different strategies to build a fuzzing approach, search-based fuzzing has been shown good empirical results for complex software such as LTL and SMT solvers [CDC$^+$24; YHT$^+$21].

Search-based fuzzing has been shown as a prominent strategy to find bugs. Search-based fuzzing generates inputs during a search process, while the inputs generated are progressively improved and evaluated by fitness functions. The search process is instantiated by a wide range of optimization algorithms that includes evolutionary and stochastic meta-heuristics [CDC$^+$24; RJK$^+$17]. The optimized fitness functions are defined in the domain of the target system. Among them, coverage and input structure validity are adopted [AZG24; CDC$^+$24].

Another prevalent manner to describe fuzzing approaches is by the term black-box fuzzing. The term describes techniques that do not require insight into the internal elements of the system under testing, such as coverage or control flow graph. Instead, black-box fuzzing is guided by the behavior of the input, output, and data. The applications include network protocols, file systems, compilers, and the operating system [MHH$^+$21].

Associated with black-box fuzzing are techniques such as differential or model-based fuzzers. Differential testing or fuzzing compares similar program. For example, different programs, versions, or configurations follow the same specification. It aims to facilitated the identification of discrepancies between the similar programs, which are likely to indicate the presence of a bug [MHH$^+$21].

In the case of model-based fuzzers, they contrain the random inputs for a specific language, or even a grammar. In such case, the model of the language in question is incorporated into the fuzzer itself. Thus, the model-based fuzzer always produce syntactically correct inputs[MHH$^+$21; SZ22]. Ideally, the generated inputs are also semantically correct or approximate, when guided by search-based approach that are optimized by semantic-based fitness .

6.3 Portfolio

It is uncommon to find the best algorithm or heuristic for a particular problem. For NP-Hard problems, heuristics perform well or poorly on different classes of inputs. In other words, the algorithm may offer the best performance on particular problem instances [LNA$^+$03; LNS02]. Especially, when the inputs are correlated, thus, algorithms that introduce a poor performance on average can be combined and consequently build a resulting algorithm with a low average runtime.

The mentioned combination of algorithms is called algorithm portfolio. The term was introduced by Huberman *et al.* [HLH97] that describes a strategy of running algorithms in parallel with the goal of obtaining a better distribution performance. For instance, Huberman *et al.* [HLH97] also verify experimentally algorithm portfolio on graph-coloring problem (NP-complete problem). In summary, the algorithms portfolio exploits the lack of correlation in the best performance of multiple algorithms to achieve better performance. Overall, it obtains an improved performance in the average case.

Associated with the algorithm portfolio is defining which algorithms to use. In this order, the algorithm selection problem consists of a method to select the best algorithm configuration for a particular problem. Rice *et al.* [Ric76] defines such as which algorithms should be run to minimize some performance objective [XHH$^+$08]. The algorithm selection problem is described itself as an NP-Hard problem [Ric76]. For this reason, many heuristics and strategies have been proposed to find the best algorithm configuration. Applications include propositional satisfiability problem [BIB22].

6.3.1 Algorithm Portfolio

There are many methods for portfolio algorithms and algorithm selection problems. In the case of algorithm portfolio, we mention:

1) *Pure Portfolios*: algorithms run in parallel and compete to solve the same problem. Xu *et al.* [XHH$^+$12] states that pure portfolio simulates the virtual best solver. It is a theoretical device that has access to a set of algorithms for a particular problem and the device selects the single algorithm that solves the problem fastest.

2) *Parallel Portfolio*: explicit divide the search space in partitions. After that, each algorithm explores a subset of partitioning. It is naturally convenient for a wide range of algorithms. Among them, algorithms for propositional satisfiability problem such as DPLL and CDCL are based on the notion of searching the space of partial variable assignments [SS24]. In this order, the parallel portfolio may instantiate each algorithm for a subset of partial variable assignments. Regarding the drawback, Hamadi *et al.* [HJS09] states the main problem as guarantees of fairness conditions. That is, the difficulty to balance between the different algorithms.

3) *Parallel portfolios with information sharing*: algorithms run in a complementary sequential algorithm. The sequence allows them to cooperate and improve their performance. In this sequence, each algorithm shares information. For instance, ManySAT [HJS09] shares clauses, lemmas, or even variables to different heuristics and algorithms such as standard DPLL algorithm, restart policies, activity-based variable selection heuristics, and clause learning. All the information is shared in a conventional shared-memory model.

6.3.2 Algorithm Selection Problem

For the algorithm selection problem, we mention:

1) *Winner-take-all*: measure algorithms runtime on a set of problem instances, then it adopts the algorithm that shows the best performance on average. The common strategy for algorithm selection is to select different performance algorithms on a set of distribution problems, then we adopt only the algorithm with the lowest average runtime [LNA$^+$03].

2) *Predictor*: it seems a perfect oracle or a heuristic approximation to the real runtime. The idea is based on building an approximate runtime predictor, in which machine learning is adopted such as empirical hardness models, which predict the computational cost of a particular combination of algorithms over a set of instances. Often, empirical hardness of individual instances or distributions of NP-Hard problems is used to algorithm selection problem [LNS02]. For example, Samulowitz and Memisevic [SM07] applied the classification to select heuristics for QBF solving during a search process.

3) *Diversity principle:* the strongest principle is the algorithm diversification. It employs different algorithms, data structures, searches, or implementations[SD11]. The main goal is to reach different behaviors across the portfolio members. For instance, an algorithm may answer faster for a particular instance, while other algorithms answer faster for other instances. The main limitation is the portfolio size [BIB22], since the number of algorithms with different design decisions may limit the portfolio benefit.

6.3.3 Linear Temporal Logic Portfolio

Many algorithms may compose an LTL portfolio from a theoretical point of view. We may present successful strategies to address satisfiability and their composite limitation: (i) Bounded Model Checking (BMC), (ii) Binary Decision Diagram (BDD), and (iii) Tableau. In what follows, we present them.

Bounded Model Checking (BMC) explores a state space based on a path segment given a length k. The initial idea is to find a counterexample of a length k. Thus, the solvers commonly translate a path segment k to a propositional formula, and the counterexample exists iff the propositional formula is satisfiable. The translation is computed in polynomial time. The main notion is to consider a bound k, in which the path is a maximal length of a counterexample. After that, the algorithm progressively increases the bound k [CBR+01; BCC+99b]. NuSMV implements a BMC algorithm [CCG+02].

BDD-based Model Checking associates elementary formulas or subformulas obtained by an input LTL formula [CGH97; CGP+02a]. Each boolean formula then is converted to a binary decision diagram (BDD) structure. The BDD maintains a compressed form that deals with a potential state-space explosion. Also, the structure instantiates common operators over the compressed form as conjunction, disjunction, and negation in polynomial time. The BDD structure generated by the boolean formulas often represents an implicit tableau that allows the solving procedure.

Tableau methods build a graph or tree shaped when the definition of a node is a set of formulas and the relations are defined by this set. Based on tableau, the LTL solving technique traverses to find suitable models for an input LTL formula. One-pass and multi-pass tableau methods have been proposed. Among them, we observe the Schwendimann method that is implemented by PLTL [Sch98a], and the Reynolds method [Rey16a] implemented by Black [GGM+21b].

Practical limitations exist in the composition of the mentioned strategies. Among the limitations, a parallel portfolio with information sharing is absent for LTL, since many different design decisions have been taken. For instance, NuSMV (BMC version) commonly translates LTL formulas to propositional formulas, while NuSMV (BDD version) integrates the mentioned binary decision diagram. The different data structures or data representations do not share compatible information during the search exploration. Even for the tableau strategy, different structures and representations are adopted. For example, black implements Reynolds' tableau that constrains the construction of repetitive loops and also integrates prune rules [Rey16a], while PLTL implements tree-shaped or graph-shaped tableau with repetitive loops. The mentioned different modifies the search and also data representation [Rey16a].

Thus, the information sharing of data during the execution is typically unreachable. Moreover, the portfolio strategy for exploring a subset partitioning is also hard since the mentioned algorithms do not share a common strategy to explore the search space. Thus, the conventional portfolio in the case of LTL is a pure portfolio, in which the different implementation strategies run in parallel.

Regarding the algorithm selection problem, predictor algorithms are unavailable for LTL solvers. Moreover, the individual performance analysis of LTL solvers suggests the best performance, when different strategies are combined rather than selecting the best average runtime [SD11]. In that sort, we adopt the diversity principle for our portfolio. In our work, we explore BMC, BDD, and different tableau strategies that are contained in the latest versions of LTL solvers.

Besides, there are other satisfiability algorithms. However, some algorithms were overperformed as presented by previous performance evaluation [LZP+15]. In particular, the automata-theoretic approach translates LTL formulas to automata. The alternating automata are commonly used to LTL satisfiability since the automata generalize the notion of non-deterministic by generating several successor states and are exponentially more succinct. However, solvers that adopt automata-theoretic approach were overperformed or new versions chosen different satisfiability algorithms [LZP+15].

6.3.4 Reliable portfolio

In this work, we propose the addition of confirmation answer. Previous studies present bugs across many versions and strategies of LTL solvers [CDC+24]. In that order, LTL portfolio may also combine and increase the presence of bugs.

For that reason, we claim the addition of confirmation answer. In other words, the reliable portfolio does not answer for the first LTL solver that concludes the execution. The reliable portfolio only answers for the first agreement between solvers. Thus, two solvers should answer and mutually confirm the answer.

In this portfolio, we extend from the pure portfolio, solvers running in parallel and competing for the fastest answer, to the reliable portfolio, when solvers run in parallel and compete for the first confirmation answer. The confirmation answer also implements a parameter for degrees of confidence. Thus, the users may decide the number of confirmation solvers. We empirically evaluate at least two solvers confirming the LTL answers.

From a practical point of view, if a first solver answers sat and the second unsat then a third solver will decide the final answer. Thus, if the third solver answers SAT, then the final answers is SAT insofar as there is a double confirmation of SAT answer. In the case of crash or error message, the reliable portfolio ignores the case.

We guide the algorithm selection problem for the diversity principle as previously recommended by performance analysis of LTL solvers [SD11]. We integrate the three mentioned strategies for the LTL satisfiability problem (BMC, BDD, and tableau). Moreover, we consider different implementations for each strategies. For instance, tableau is implemented twice for Black and also PLTL. In the first implementation, Black implements Reynold's tableau, while PLTL implements a multi-pass search algorithm for a conventional LTL tableau [Rey16a].

For the empirical evaluation, we explore a fuzzing approach to check the presence of bugs in a traditional pure portfolio, and also the reliable portfolio with double confirmation answer. After that, we consider the performance of the reliable portfolio compared to a single LTL solver.

6.4 Fuzzing

We select SpecBCFuzz to generate random LTL formulas for testing solvers. The decision was driven based on the previous capability to find bugs in LTL solvers. SpecBCFuzz found bugs in 4 solvers and 18 out of 21 configurations and versions. In total, SpecBCFuzz found 16 bugs in LTL solvers. Among them, we can mention soundness, flaky, performance, and crash bugs. Moreover, the empirical results also report that SpecBCFuzz outperforms a probabilistic grammar fuzz [CDC$^+$24]. Therefore, we can test the different portfolios (pure and reliable) based on SpecBCFuzz. Overall, SpecBCFuzz is a search-based fuzzing. That is, SpecBCFuzz generates and evolves LTL formulas in a search process, while off-springs are evaluated by the fitness functions.

Firstly, an LTL specification and a set of boundary conditions (divergent cases) are provided as input. The LTL specification and their boundary conditions work as seeds. The LTL specification represents a common domain insofar as the specifications are written in the format of goals and domain properties, and they are representative cases of LTL constraints that LTL solvers should compute satisfiability. Moreover, boundary conditions represent a conflict that results in the loss of satisfaction of the mentioned goals and domain properties. Boundary conditions are seeds extensively explored in the search process, while LTL specifications are bootstraps and guides for the search.

The search process focuses on evolving LTL specification into LTL repairs that the boundary conditions (divergent cases) do not apply loss of satisfaction. Moreover, SpecBCFuzz also applies constraints in the search space in terms of the semantic and syntax distance of the original LTL specification. Finally, SpecBCFuzz also evaluates continually the satisfiability of the LTL repair produced in the off-springs. In total, SpecBCFuzz evaluates four fitness functions in the search.

Therefore, SpecBCFuzz implements a multi-objective search algorithm based on a genetic algorithm. In that sort, SpecBCFuzz implements mutation and crossover operators to continually introduce syntax variations in the specifications. After that, the repair candidates are evaluated by the mentioned four fitness functions: (i) satisfiability of the repair, (ii) score of removed boundary conditions, (iii) semantic similarity, and (iv) syntax similarity. For the satisfiability of the repair and the score of removed boundary conditions, LTL solvers are requested, while semantic distance relies on a model counting heuristic and syntax distance in the Levenshtein distance. In summary, the LTL solvers should answer for complex and realistic inputs about the satisfiability of the LTL repair, while the set of boundary conditions are checked in terms of the permanence of the conflict or not. The boundary condition as a search objective forces the search to explore a wide spectrum characterized by the divergences from the UNSAT to SAT answers. In other words, the continuous verification forces the evaluation of LTL formulas within the boundaries of UNSAT and SAT.

Finally, SpecBCFuzz conducts a differential fuzzing, since different LTL solvers are sequentially called by SpecBCFuzz for all the LTL repair produced in the off-spring and the boundary conditions verification. The differential fuzzing result is manually analyzed for divergent outcomes. For instance, a majority of the solvers answered SAT and a single solver answered UNSAT. The mentioned pattern may indicate a soundness bug. Additional patterns are also analyzed. Among them, error messages and timeout patterns are based on the type of algorithm adopted by a subset of solvers.

6.5 Families of LTL Benchmarks

We choose the Schuppan and Darmawan families of LTL benchmarks [SD11]. The benchmarks extensively evaluate LTL satisfiability in terms of performance. It also includes previous benchmarks and adds new sets of benchmarks. In what follows, we describe them and show the range of origins, communities, and specifications.

The families of LTL benchmarks contain Rozier benchmark [RV10] that was build to evaluate the LTL satisfiability via a reduction to model checking by large LTL formulas. The experiment evaluates the performance of LTL translation tools to automata as well as symbolic and explicit model checking. The main goal is the challenge with large formulas and state spaces. For that reason, the benchmark was built on top of the randomly generated, counter and scalable patterns formula. Random formulas varying the length and number of variables. It also contains a random probability to select a temporal operator. Counter formulas represent an n-bit counter when n will vary and stable LTL properties need to hold. Scalable patterns are defined by conjunction or disjunction of pre-defined LTL properties such as safety properties.

Anzu benchmark [BGJ^{+}07] contains a generalized buffer and an arbiter for amba. They are representative industrial examples from realistic specifications, while Acacia benchmark [FJR09] presents cases based on window screens, arbiters, and traffic light controllers. Forobots benchmark [BDF09] includes specifications of robotic behaviors and his extension to many robots, food for collection, and additional constraints.

Moreover, Alaska benchmark [DDM^{+}08] introduces parametric specification of a lift system with n floors. First, a specification with a linear number of variables per floor. Second, a specification with several variables that is logarithmic in terms of number of floors. Moreover, a mutual exclusion protocol that describes liveness properties is also part of the benchmark. TRP benchmark [HS02] benchmark includes random formulas from fixed conjunctive normal form.

Furthermore, Schuppan and Darmawan also scaled up the mentioned benchmarks [SD11]. They extend sets of assumptions and guarantees with new patterns of conjunction and variants with liveness properties. In addition, they also create new benchmarks that contain patterns that explore exponential behavior and temporal formulation of the pigeonhole principle.

In general, the aforementioned families of benchmarks guide the performance evaluation for many cases, properties, and space complexity of LTL formulas. Moreover, the benchmarks are also representative of different communities [SD11], while the Rozier benchmark was created to evaluate model checking and automata transformation, for formulas in the Anzu benchmark, it presents industrial and realistic specifications. The final result is a large data set of LTL formulas that effectively guides a performance comparison or solver competition.

6.6 Empirical Study

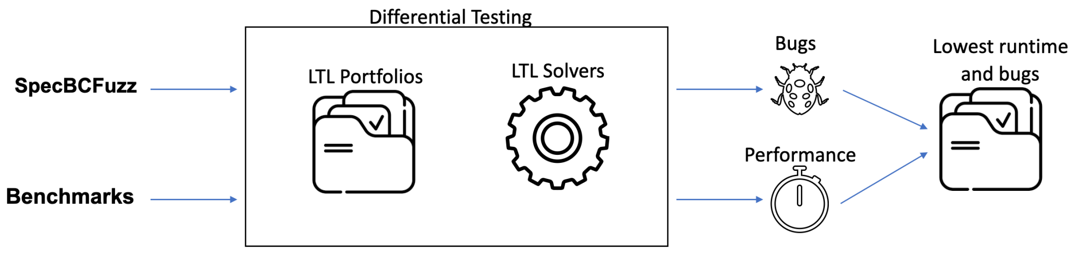


Figure 6.1: Exploring correctness and performance for LTL solvers and portfolio

Our experiment fill the lacks about performance and testing of portfolios and solvers of LTL. Previous works assessed LTL solvers in terms of time or score performance [RV10; SD11], while other studies assess them in terms of correctness [CDC$^+$24]. The same status occurs when we consider LTL portfolios. That is, the work evaluates isolated performance [LPZ$^+$13], while correctness is a common gap. Moreover, the association of performance and correctness is a common gap for both cases.

Therefore, we propose new empirical studies when single LTL solver and LTL portfolios are evaluated and compared in terms of correctness and performance at the same time. Figure 6.1 represents it. Moreover, the empirical results point to a new reliable portfolio, since it reduces the presence of bugs, and it also presents the lowest runtime when compared to a single LTL solver.

Thus, we analyze and empirically evaluate pure and reliable portfolio in terms of soundness and performance:

- **RQ1:** What is the occurrence of bugs in a pure portfolio of LTL solvers?
- **RQ2:** What is the occurrence of bugs in a reliable portfolio of LTL solvers?
- **RQ3:** How effective is the performance of the reliable portfolio?

Regarding RQ1, we run SpecBCFuzz against a pure portfolio of LTL solvers. After that, we manually evaluate the outcomes of differential fuzzing against a single solver NuSMV (BDD version). Based on the inconsistency patterns. For instance, conflict answers (SAT and UNSAT), error message, and even particular timeouts. The result is a set of reproducible bugs and their types. We also compare them to a previous works.

To answer RQ2, we run SpecBCFuzz against a reliable portfolio. That means a set of LTL solvers running in parallel and only answers when at least two LTL solvers confirm the same answer. Based on the patterns of the differential fuzzing against a single LTL solver, we try to identify and categorize the presence of bugs. The research questions aims to evaluate in what level of confirmation the differential fuzzing is not more able to identify bugs.

RQ3 evaluates the performance of reliable portfolio. For that reason, we run reliable portfolio with two confirmation answers against a single LTL solver, and based on a quantitative comparison we evaluate the performance is terms of SAT and UNSAT answers and also the presence of timeouts in the benchmark execution.

In what follows, we introduce the LTL solvers, fuzzing, benchmark, and experimental settings.

6.6.1 Experimental Instruments

We selected four LTL solvers and six configurations for the least releases. The selected solvers are state-of-the-art for LTL satisfiability in different terms such as performance, model checking tools, expressiveness, and extensibility [CDC$^+$24; SD11]. The criteria also maintains diversity principle such as three different algorithms are presents in the solvers. Moreover, different implementations and design decision are also present per each configuration. In what follows, we describe the solvers, algorithms, data representation, and configurations.

Aalta works on the fly and builds a satisfying LTL formula without constructing a full state representation. Aalta version 2 [LZP$^+$15] implements a depth-first search (DFS) responsible to build a temporal transition system and search for a satisfiable trace. Specifically, the DFS creates an initial state and its successors using SAT techniques. The original formula is converted to an equivalent next normal form, and the conversion puts subformulas in terms of the operators Until and Release in the scope of the operator Next. In this way, the recursive functions are created and form the current state (propositional formula) and their successors (subformulas composed by the operator next) as mentioned.

BLACK [GGM21; GGM$^+$21b] implements a one-pass tree-shaped tableau developed by Reynolds [Rey16a] for satisfying LTL formulas. Essentially, BLACK generates a tableau based on an LTL formula provided in the input. After that, BLACK converts the tableau for a depth k in a propositional formula based on the pruning rule. Interactively, the value of k increases until the search finds an accepted branch in the tableau or a witness of unsatisfiability is detected. The authors claim the completeness of the algorithm [GGM+21b] since it follows the pruning rule of Reynolds'tableau [GGM+21b; Rey16a]. In our experiment, the propositional formulas are evaluated by Z3 [dMB08].

PLTL widely integrates different algorithms. PLTL released an implementation of Wolper's method [Wol85], the tableau method as well. The algorithm starts by building a cyclic graph. After that, the algorithm conducts multiple passes, while pruning inconsistent nodes and nodes that contain unfulfilled eventualities. Furthermore, PLTL also contains a BDD-based version. The algorithm builds subsets of the Fischer-Ladner closure [FL79] of an input formula. The algorithm then uses the greatest fixpoint approach [Mar05] to progressively delete the mentioned sets by the SAT semantic. In the empirical evaluation, we consider the multi passes and BDD version.

NuSMV implements BDD-based model checking and also explores SAT-based model checking [CGP+02b]. NUSMV version 1 was implemented upon the BDD structure. The BDD compresses formulas and subformulas obtained by the input LTL formula to deal with the state-space explosion. The BDD structure generates an implicit tableau and turns available traditional logic operators (e.g., conjunction and disjunction), where the implicit tableau enables support for the solving. NuSMV version 2 constructs an internal representation of the model based on a simplified version of the Reduced Boolean Circuit (RBC) [CCG+02]. The main notion of the bounded model checking and the RBC representation is to consider counterexamples of a particular length k and a finite prefix of a path. After that, the algorithm converts to a propositional formula, which is satisfiable iff such a counterexample exists. This constraint allows the algorithm to identify a counterexample without passing the entire search space. To improve the performance, the algorithm often includes splitting heuristics to quickly identify the counterexample [BCC+99a]. In the experiment, we run both version. Previous empirical investigations have been unable to identify any bugs in the NuSMV (bdd version) [CDC+24; SD11]. Consequently, we have elected to adopt NuSMV (bdd version) as the single robust LTL solver in the empirical evaluation.

6.6.2 Experimental Settings

Fuzzing depends on many parameters. In that way, we follow Carvalho *et al.* [CDC+24] parameters. Regarding the seeds, we maintain 25 formal requirements specifications extracted from the literature and different benchmarks [CDC+24; DCA+18; DMR+18; DRA+16; LWS+21; vLDL98b]. Moreover, the boundary conditions of each specification is also a seed. The boundary conditions were identified automatically in pre-processing. For the search step, the population size of 100 individuals was defined and the fitness evaluation was limited to a number of 1000 individuals. The probability of crossover application was 0.1, while mutation operators were always applied. Moreover, the timeout of the model counting as 30 seconds.

Benchmarks build by Schuppan and Darmawan [SD11] contain statics LTL formulas in SMV format. We convert them to a generic LTL format supported by OwL library [KMS18] to parse and manipulate the LTL specifications insofar as each LTL solver presents its input constraints.

Experimental execution run SpecBCFuzz and Schuppan's benchmark on a Xeon E5 2.4GHz and 64GB of memory. The operating system is Redhat Linux, version 8.10. Moreover, we also set two days as a time threshold per the entire execution of each job. In total, we launched 25 execution. In the case of SpecBCFuzz, we provide a different specification and its associated boundary conditions set per each job. Furthermore, we conducted the experiment process 10 times to reduce random elections of the search algorithm. Moreover, LTL solvers were configured with a timeout of 300 seconds. In the case of Schuppan's benchmark, we launched a execution per each sub-families. In total 333 executions. Regarding the execution, we run once the mentioned benchmarks since randomness behavior is not a potential bias. The timeout for each formula and each LTL solver or portfolio is 60 seconds.

Statistics For evaluation, our experiments are based on non-parametric tests and ratio comparison. We perform a performance evaluation in two parts. First, we compare the runtime performance. In that manner, we performed statistical analysis, that is, the Wilcoxon signed rank test [Wil45] to compare the single robust LTL solver and the reliable portfolio strategy. Moreover, we also compute the odds ratio to evaluate the proportion of timeouts [RLS04].

6.7 Results

6.7.1 Pure Portfolio

We build a pure portfolio with six LTL solvers. They are Aalta (version 2), Black (Z3 SAT solver), NuSMV BMC and BDD, PLTL tableau (multi-pass algorithm) and BDD. They run in parallel at the same time and compete for the fastest answer. Thus, as soon as the first answer is found, the pure portfolio answer SAT or UNSAT.

In order to test the pure portfolio, we run the fuzzing SpecBCFuzz. As mentioned, SpecBCFuzz is a differential fuzzing. Thus, we also execute NuSMV (BDD version) as a baseline since previous studies report it as potentially absent of bugs [CDC$^+$24]. Table 6.2 presents the bugs found. All the bugs for the pure portfolio found by SpecBCFuzz are soundness bug. That is, the solver answer SAT when the correct answer would be UNSAT and the opposite. In total, there are three bugs.

Overall, pure portfolio combines soundness bugs from each solver. The bugs come from Aalta, and PLTL, while Black and NuSMV does not contain bugs for

Table 6.1: Bugs found in the Pure Portfolio

Bug Type	**Bug Description**
Wrong answer	Operator G and the negation of its operand always answer SAT
Wrong answer and Flaky	Presents a wrong answers time to time
Wrong answer	Answer differ from the baseline for long associations of G, X, F, and U operators

the mentioned portfolio. The reason for Black and NuSMV does not present bugs in the pure portfolios is associated to a potential absence of wrong answers bugs.

Hang loop, crashes, and exception throws are not common for pure portfolio. The reason is the fast answer of the first solver avoid longs execution. Consequently, the performance bugs mentioned are unlikely to happen.

6.7.2 Reliable Portfolio

We build a reliable portfolio with the same six LTL solvers. The LTL solvers run in parallel as a pure portfolio. However, the answer is not defined by the fastest solver. In fact, the answer is defined by the agreement of the first pair of LTL solvers once the agreements is SAT or UNSAT.

We conduct a differential fuzzing with SpecBCFuzz, then we run reliable portfolio and NuSMV (bdd version) as a baseline comparison since previous studies were not able to present bugs in this particular solver version. We run SpecBCFuzz and the empirical result does not present bugs. The simple addition of double confirmation of answers was enough to mitigate the previously mentioned bugs in the pure portfolio as well keep the performance and crash bugs absents, even that they are present in part of the solvers that form the reliable portfolio.

Furthermore, we also compute the presence of performance warnings, in which associates timeout warning in a set of one or more solvers. In total, the empirical evaluation does not reveals performance warning. That is, the addition of double answer confirmation does not introduce any timeout for the reliable portfolio and the default timeout parameter of 300 seconds.

6.7.3 Runtime Performance

To answer RQ3, we conduct two hypothesis tests. The first test compares the satisfiable and unsatisfiable answers for the reliable portfolio and NuSMV (bdd version), while the second test compares the timeouts per each mentioned side.

In the first test, the null hypothesis states that the reliable portfolio answers are equal to NuSMV. The alternative hypothesis states that reliable portfolios runtime average is lower than the NuSMV. We assess both sides in terms of the families of benchmarks [SD11] and their runtime performance.

Figure 6.2 presents the result. Both median are flattened in less than 1 second. The plot suggests that both solvers are able to efficiently answer many LTL formulas in less than a few seconds. Moreover, the plot shows a large spread in NuSMV from 1 to 60 seconds. In the case of reliable portfolio, it is concentrated from 1 to 10 seconds, while a few formulas are answered from 10 to 35 seconds.

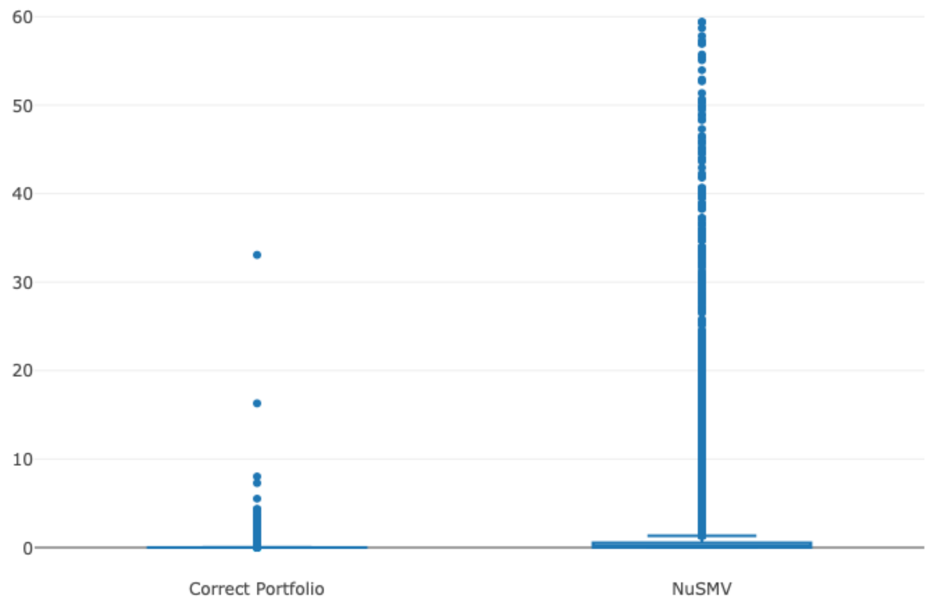


Figure 6.2: Box plot of runtime (seconds) for satisfiable and unsatisfiable answers for Reliable portfolio and NuSMV (bdd version).

In order to assess the null hypothesis, we run Wilconxon non-parametric test. $p-$value is lower than 0.01. Thus, we reject the null hypothesis and conclude that reliable portfolio presents lowest runtime average than an NuSMV. In fact, the result points out that include a answer confirmation can reduces the runtime performance. However, the average runtime continues lowest than run a single robust LTL solvers, while robustness and correctness non-functional requirements are preserved as shown in RQ2.

Furthermore, we also assess the timeout. Table 6.2 presents the timeout cases for reliable portfolio and NuSMV. In the second statistic test, the null hypothesis

Table 6.2: Timeout for reliable portfolio and NuSMV (bdd version)

	Answers	**Timeout**
Reliable portfolio	6819	361
NuSMV (bdd version)	5920	1260

states that there is no change in reliable portfolio and NuSMV timeouts. For the alternative hypothesis, the reliable portfolios timeout are lowers than the NuSMV.

We compute odds ratio to compare the different ratio of timeouts. The statistical test presents a odds ratio of 0.25. The confidence intervals is from 0.21 to 0.29 for $\alpha = 0.01$. Thus, we reject the null hypothesis since the confidence interval is lower than one.

Therefore, we conclude that introduce confirmation answer in a pure portfolio does not reduce it to the runtime performance of a single robust LTL solver. Contrary to that, the reliable portfolio answers fast than a single robust LTL solver and produce less timeouts.

6.8 Threats to Validity

The following section addresses threats to the validity of the empirical study. One potential threat to the veracity of our study is the use of third-party tools. However, SpecBCFuzz is designed to rely on reliable frameworks and libraries. With regard to optimization procedures, SpecBCFuzz incorporates the standard JMetal framework [NDV15] and runs NSGA-III [DJ14]. Schuppan's benchmark is a static families of benchmarks and does not depend on third-party tools. For our study, the LTL formulas are handled by the OwL library [KMS18].

The presence of bugs in these components may affect the empirical results. For example, they could lead to the generation of false positive evidence for a potential bug. To address these potential threats to validity, we conducted a manual double-check of the differential results. In addition, all libraries and frameworks have been extensively used in various research studies, and the results remain consistent.

We remove the generalized buffer and the scaled-up formulas from the anzu benchmark since the LTL formulas are not well-structured in a format that the OwL library could read. In general, the families of benchmarks is a large dataset of LTL formulas and the exclusion should not cause any niggles.

The external validity of the experiment is diminished due to the utilization of multiple solvers, data structures, and LTL satisfiability algorithms. In addition, inputs or seeds were extracted from a diverse range of specifications and sources [CDC+24], while the boundary condition was extracted from multiple executions. To mitigate the reproducibility of the experiment, the tools were run multiple times, since they contain random components.

6.9 Conclusion

LTL solvers compete for runtime performance and applications. Thus, it is uncommon to find the best solver for LTL satisfiability. Portfolio proposes to combine LTL and run them in parallel. The resulting portfolio has runtime performance that is lower than that of a single LTL solver. However, previous work presents bugs in LTL solver for many different types of algorithms and heuristics [SD11; CDC+24]. Therefore, the combination of LTL solvers can also combine and preserve bugs in the first answer strategy. In fact, our experiment shows the presence of previously known bugs in single LTL solvers and also in the pure portfolio. Thus, we propose a new design decision for the LTL portfolio, called the reliable portfolio.

The reliable portfolio is based on common strategies and new strategies to reduce the presence of bugs in the resulting portfolio. Among them, the diversity principle states algorithm diversification that employs different algorithms, data structures, searches, or implementations. It aims to reach different behaviors. Thus, each portfolio member takes advantage of the best time performance for a particular subset of a given problem. Moreover, we modify portfolio decisions to introduce a confirmation answer. That is, the portfolio does not answer for the fastest answer. However, it answers for the first confirmation answer under a given confirmation level. That is, the portfolio replicates the answer when at least two solvers accept an agreement for a given input.

Therefore, we conduct a two folds study to empirically evaluate reliable portfolio. First, we evaluate the correctness in terms of a differential fuzzing approach called SpecBCFuzz. Second, we evaluate the performance degradation caused by the confirmation answer in terms of families of benchmarks.

SpecBCFuzz is a search-based fuzzing. Thus, SpecBCFuzz evolves LTL formulas in a search, while their offspring are evaluated by fitness functions. SpecBCFuzz receives as input a set of seeds built by formal requirement written in LTL and a set of divergence cases for the mentioned formal requirement, named boundary conditions. In summary, SpecBCFuzz produces complex inputs for the LTL solvers. The differential fuzzing result does not present new bugs for a reliable portfolio that answer for two confirmation answer. Moreover, the reliable portfolio is absent from the timeout answer during the SpecBCFuzz evaluation.

We evaluated the reliable portfolio and a single robust LTL solver in terms of the benchmark families [SD11], their runtime performance, and the occurrence of the timeout. Our empirical evaluation presents that the reliable portfolio runtime average is lower than the a single robust LTL solver represented by NuSMV (bdd version). In addition, we also evaluate the timeout. The reliable portfolios timeout are lower than the single robust LTL solver. Therefore, we conclude that the confirmation answer in a pure portfolio does not reduce it to the runtime performance of a single robust LTL solver. In contrast, the reliable portfolio responds faster than a single robust LTL solver and produces fewer timeouts.

7

Conclusion

This chapter presents the overall conclusion of the dissertation and proposes potential research directions.

Contents

7.1 **Summary of Contributions** 128
7.2 **Perspectives** 130

The main objective of this dissertation was to enhance the formal specifications written in linear temporal logic (LTL). In addition, we also enhance their satisfiability tools (LTL solvers). Conflicts or divergences in formal specification may be caused by different sources. Among them, stakeholders or even engineers may propose conflict goals for the same software-to-be. It usually happens to a myopic vision or a selfish perspective that does not allow one to compare the current goal with the entire software-to-be specification. In that manner, we proposed an automated tool to repair the specification and significantly reduce the presence of conflicts characterized by boundary conditions. Moreover, we extend our empirical experiment to evaluate LTL solvers. We interpret and adjust our search-based software engineering to the fuzzing context. From then on, we are able to effectively identify bugs in LTL solvers and outperform the state-of-the-art. In what follows, we discuss the summary of contributions about the aforementioned contributions and consider future direction.

7.1 Summary of Contributions

Goal-conflict resolution is a key step in goal-oriented requirements engineering (GORE). GORE methodologies use a logical formalism to specify the software-to-be behavior. In Chapter 3, we introduce ACoRe, the inaugural automated methodology for goal-conflict resolution. In general, ACoRe implements a search-based approach to software engineering, whereby it considers a set of previously identified conflicts expressed in LTL and computes a set of repairs that effectively remove such conflicts. Among the aforementioned objectives, ACoRe has been designed with the objective of reducing the syntactic distance and increasing the semantic similarity between the original specification and the proposed repairs. ACoRe was evaluated in accordance with realistic specifications. The results demonstrated that the genetic algorithms tend to generate a greater number of non-dominated repairs. The evaluation demonstrated that genetic algorithms typically yield a greater number of non-dominated repairs. Furthermore, ACoRe produces a sufficient quantity of repairs per specification, thereby facilitating the engineer's ability to analyze and select the optimal repair. Furthermore, it is evident that the genetic algorithms demonstrate superior performance, as evidenced by several evaluation metrics, including the number of repairs, the number of repairs that do not introduce new conflicts, the number of repairs that resemble manually written fixes, and standard quality indicators for multi-objective algorithms.

LTL Specification Mining. There are numerous applications of LTL in formal methods. In order to facilitate the specification of a target system, a recent research field concerning LTL is the mining of specifications from common software representation. The mining problem has been partially addressed by various approaches, including pattern matching and constraint solver techniques. The LTL mining problems are related to the mining of LTL formulas from provided artifacts. Common software representations are finite traces and templates that represent a target system. The results of the experiment indicate a lack of alignment between the LTL properties mined and a comparable original set of LTL specifications extracted from the literature. Furthermore, we evaluate the effectiveness and capabilities of LTL mining tools, which cover an original set of specification properties written in LTL. We evaluate the precision of the mined properties in relation to the covered original properties. The results indicate that the mined properties are capable of covering the original properties in a few cases. In summary, the precision of the LTL mining tools is very low, making the active process time consuming and error prone.

Finding bugs in LTL solvers. LTL solvers are constructed using a multitude of sophisticated satisfiability algorithms. For this reason, we develop SpecBCFuzz as a search-based fuzzing approach that has demonstrated efficacy in identifying pertinent failure-inducing inputs. This is accomplished through the optimization and search of SpecBCFuzz using boundary conditions, syntactic and semantic similarity, and general satisfiability of the modified LTL specifications, as presented in Chapter 4. Furthermore, a comparative analysis was conducted to evaluate the empirical results generated by SpecBCFuzz. The objective of this analysis was to determine whether the fuzzing approach itself is capable of producing similar outcomes when the fitness functions and seeds are disabled. Subsets of fuzzing strategies were used to assess the bias present in the final result of the tool. This was achieved through the implementation of an unguided search conducted in the presence of realistic specifications. Furthermore, the semantic and syntax fitness functions were deactivated to evaluate the influence of boundary condition fitness in the context of previously evaluated seeds. Empirical findings indicate that reducing the fitness function also decreases the capacity of SpecBCFuzz. Overall, the empirical evaluation identified 16 bugs in the LTL solvers under analysis. In addition, warning patterns were identified, which are particularly useful for performance testing purposes, as they demonstrate performance issues in a specific LTL solver, version, or even an implemented algorithm.

Reliable LTL portfolio. The combination of LTL solvers can result in the coexistence and perpetuation of bugs inherent to the first answer strategy. Our experimental findings suggest the presence of previously identified bugs in individual LTL solvers and within the pure portfolio, as detailed in Chapter 5. In light of these considerations, we propose a novel design decision for the LTL portfolio, which we coin the term reliable portfolio. The reliable portfolio is based on a combination of established and novel strategies designed to minimize the prevalence of bugs in the resulting portfolio. Among these strategies is the diversity principle, which involves the diversification of algorithms through the employment of different algorithms, data structures, searches, or implementations. It achieves a variety of results. Consequently, each portfolio member capitalizes on the optimal time performance for a specific subset of a given problem. In addition, modifications are made to the portfolio decisions to incorporate a confirmation answer. In other words, the portfolio does not respond to the fastest answer. Nevertheless, it responds to the initial confirmation query at a specified confirmation level. In other words, the portfolio replicates the answer when at least two solvers agree on a given input. Our empirical evaluation demonstrates that the reliable portfolio runtime average is less than that of a single robust LTL solver. It can be concluded that the confirmation answer in a pure portfolio does not reduce it to the runtime performance of a single robust LTL solver. In contrast, the reliable portfolio responds more rapidly than a single robust LTL solver and produces a smaller number of timeouts. The differential fuzzing result does not present new bugs for a reliable portfolio that answers with two confirmation answers. Furthermore, the reliable portfolio is absent from the timeout answer during the SpecBCFuzz evaluation.

7.2 Perspectives

Propositional satisfiability (SAT) and quantified boolean formulas (QBF) solvers. SpecBCFuzz is coupled to the LTL grammar and the definition of boundary conditions that are also under the LTL abstraction since the goals are written in LTL for the GORE methodology. Our intention in the future is to build an extensive tool. In other words, a differential search-based fuzzing that adopts reliable criteria such as syntax and semantic similarity to constrain the search space and maintains the original seeds realistic and complex to indicate failure-inducing inputs. Thus, future differential search-based fuzzing includes additional grammar, such as propositional logic and quantified boolean formulas. Moreover, the relevant concept of boundary condition should also be extended to the aforementioned logic. The final result is a new fuzzing with the capability to test additional solvers, such as the SAT and QBF solvers. Furthermore, the reliable portfolio experiment may be extended to SAT and QBF portfolios.

Binary Decision Diagram (BDD). Many data structures and search algorithms can be useful in the implementation of LTL, SAT, or QBF solvers. Among them, BDD represents one of the most solid data structures for this task. In our previous experiment (Chapters 4 and 5), the most robust LTL solver was implemented by converting the LTL formulas into the SAT formula and representing them in BDD. However, many implementation details exist, such as the fact that many tools to deal with BDD are available in the literature. They may include different interfaces and operators for BDD. Therefore, we intend to develop a new fuzzing approach to systematically test BDDs. In that manner, we intend to build a new fuzzing approach that deals with the SAT grammar complexity and the

BDDs interface variants to identify bugs.

List of publications and tools

Papers published

- Luiz Carvalho et al. Acore: automated goal-conflict resolution. In Leen Lambers and Sebastián Uchitel, editors, *Fundamental Approaches to Software Engineering*, pages 3–25, Cham. Springer Nature Switzerland, 2023
- Luiz Carvalho et al. Specbcfuzz: fuzzing ltl solvers with boundary conditions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, Lisbon, Portugal. Association for Computing Machinery, 2024

Papers submitted

- Luiz Carvalho *et al.* Automated Goal-Conflict Resolution. *Formal Methods in System Design, FMSD*
- Luiz Carvalho *et al.* Evaluating Specification Mining in Linear Temporal Logic. *International Symposium on Software Reliability Engineering, ISSRE'25*
- Luiz Carvalho *et al.* Fuzzing Linear Temporal Logic Solvers. *Transactions on Software Engineering and Methodology, TOSEM*
- Luiz Carvalho *et al.* Fuzzing Linear Temporal Logic Portfolio. *International Conference on Software Testing, Verification and Validation, ICST'25*

Tools included in the dissertation

- ACoRe: Automated Conflict Resolution
  - `https://github.com/lzmths/acore`
- SpecBCFuzz: Specification and Boundary Conditions Fuzzing.
  - `https://github.com/lzmths/SpecBCFuzz`

List of figures

3.1 Syntactic and semantic similarities between the original specification and the manually developed repairs proposed by Letier [Let01] (ground-truth). 32
3.2 Mine Pump Controller: Syntactic and semantic similarities between the ground-truth repairs and the ones generated by ACoRe w.r.t. the original specification. 34
3.3 Overview of ACoRe. 35
3.4 Examples of the mutations that ACoRe implements. 39
3.5 Examples of the combinations that ACoRe implements. 40
3.6 Boxplot that summarizes the HV of the Pareto-optimal sets generated by ACoRe when uses different search algorithm (higher value is the better). 49
3.7 Boxplot that summarizes the IGD of the Pareto-optimal sets generated by ACoRe when uses different search algorithm (lower values is the better). 50

4.1 Comparison of mined specifications and ground truth of specifications 65
4.2 Box plot of precision for Texada. 69

5.1 Exploring the vicinity of the divergences. 82
5.2 Evolutionary search implemented by SpecBCFuzz. 83
5.3 Mutation and Crossover operators. 88

6.1 Exploring correctness and performance for LTL solvers and portfolio 118
6.2 Box plot of runtime (seconds) for satisfiable and unsatisfiable answers for Reliable portfolio and NuSMV (bdd version). 123

List of tables

3.1 LTL Requirements Specifications and Goal-conflicts Identified. . . . 42
3.2 Effectiveness of ACoRe in producing repairs. Column "#repairs" represent the average number of repairs in the Pareto-optimal set produced by the search algorithms. 45
3.3 ACoRe effectiveness in producing a exact match with the ground-truth 46
3.4 HV and IGD between the closest repairs produced by ACoRe to the ground-truth 48
3.5 HV and IGD assessment by ACoRe 51

4.1 Seeded LTL formulas and boundary conditions. 66
4.2 Mined LTL properties and their matches and implications 68

5.1 Solvers and versions used. 91
5.2 Seeded LTL formulas and boundary conditions. 93
5.3 Bugs found in the LTL solvers 95
5.4 Inconsistency patterns associated with bugs 96
5.5 Bugs fixed, confirmed by theoretical properties, or confirmed by developers. 97
5.6 Sørensen–Dice coefficient of warning patterns 98
5.7 Patterns of inconsistency and their occurrence described by minimum, maximum, average, and effect size ($\hat{A}_{12}$) 100

6.1 Bugs found in the Pure Portfolio 122
6.2 Timeout for reliable portfolio and NuSMV (bdd version) 124

Bibliography

[AB11] 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, pages 1–10, Waikiki, Honolulu, HI, USA. Association for Computing Machinery, 2011 (cited on pages 43, 51).

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010 (cited on page 12).

[ACvL20] Dalal Alrajeh, Antoine Cailliau, and Axel van Lamsweerde. Adapting requirements models to varying environments. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020, Seoul, South Korea, May 23-29, 2020*, 2020 (cited on page 13).

[AGS19] Moumita Asad, Kishan Kumar Ganguly, and Kazi Sakib. Impact analysis of syntactic and semantic similarities on patch prioritization in automated program repair. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 328–332, 2019 (cited on page 37).

[AIL$^+$07] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive systems: modelling, specification and verification*. Cambridge University Press, 2007 (cited on pages 12, 52).

[AKR$^+$09] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastin Uchitel. Learning operational requirements from goal models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 265–275, Washington, DC, USA. IEEE Computer Society, 2009 (cited on pages 12, 23).

[AMT13] 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 (cited on pages 13, 41, 64, 92).

[AZG24] Andrea Arcuri, Man Zhang, and Juan Galeotti. Advanced white-box heuristics for search-based fuzzing of rest apis. *ACM Trans. Softw. Eng. Methodol.*, 33(6), June 2024 (cited on pages 80, 111).

[BBB$^+$22] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 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. Springer International Publishing, 2022 (cited on pages 17, 91).

[BBC$^+$00] Nikolaj Bjørner, Anca Browne, Michael Colón, Bernd Finkbeiner, Zohar Manna, Henny Sipma, and Tomás E. Uribe. Verifying temporal properties of reactive systems: A step tutorial. *Formal Methods in System Design*, 16(3):227–270, 2000 (cited on page 12).

[BCC$^+$99a] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 317–320, New Orleans, Louisiana, USA. Association for Computing Machinery, 1999 (cited on pages 92, 120).

[BCC$^+$99b] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, Berlin, Heidelberg. Springer-Verlag, 1999 (cited on pages 26, 78, 114).

[BCP$^+$23] 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 2023, Lisbon, Portugal, July 15-19, 2023*, pages 1499–1507. ACM, 2023 (cited on pages 13, 14, 24, 26, 86).

[BDF09] Abdelkader Behdenna, Clare Dixon, and Michael Fisher. Deductive verification of simple foraging robotic behaviours. *International Journal of Intelligent Computing and Cybernetics*, 2:604–643, November 2009 (cited on page 117).

[Ben08] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer London, 1st edition, 2008 (cited on page 2).

[BFF$^+$18] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. *Introduction to runtime verification.* In *Lectures on Runtime Verification: Introductory and Advanced Topics.* Ezio Bartocci and Yliès Falcone, editors. Springer International Publishing, Cham, 2018, pages 1–33 (cited on page 2).

[BGJ$^+$07] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: hardware from psl. *Electronic Notes in Theoretical Computer Science*, 190(4):3–16, 2007. Proceedings of the Workshop on Compiler Optimization meets Compiler Verification (COCV 2007) (cited on page 117).

[BH08] Tevfik Bultan and Constance L. Heitmeyer. Applying infinite state model checking and other analysis techniques to tabular requirements specifications of safety-critical systems. *Design Automation for Embedded Systems*, 12(1-2):97–137, 2008 (cited on page 12).

[BIB22] Jakob Bach, Markus Iser, and Klemens Böhm. A comprehensive study of k-portfolios of recent sat solvers. In *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022). Hrsg.: Kuldeep S. Meel.* 25th International Conference on Theory and Applications of Satisfiability Testing. SAT 2022 (Haifa, Israel, August 2–5, 2022), volume 236 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 2:1–2:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (LZI), 2022 (cited on pages 112, 113).

[Bie09] A. Biere. *Handbook of Satisfiability.* Frontiers in artificial intelligence and applications. IOS Press, 2009 (cited on pages 58, 63).

[Bie21] Armin Biere. Bounded model checking. *Adv. Comput.*, 58:117–148, 2021 (cited on pages 6, 74, 76).

[BKL08] C. Baier, J.P. Katoen, and K.G. Larsen. *Principles of Model Checking.* MIT Press, 2008 (cited on page 3).

[BLB10] 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. Springer Berlin Heidelberg, 2010 (cited on pages 6, 7, 16, 18, 74, 108).

[BM20] Alexandra Bugariu and Peter Müller. Automatically testing string solvers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE ’20, pages 1459–1470, Seoul, South Korea. Association for Computing Machinery, 2020 (cited on page 17).

[BMA+22] 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), July 2022 (cited on pages 79, 110).

[BMB+18] 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. Springer International Publishing, 2018 (cited on pages 6, 7, 16, 18, 74, 108).

[BPN+17] 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, pages 2329–2344, Dallas, Texas, USA. Association for Computing Machinery, 2017 (cited on pages 79, 111).

[BPR19] 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 (cited on pages 79, 111).

[BSM+08] Sanghamitra Bandyopadhyay, Sriparna Saha, Ujjwal Maulik, and Kalyanmoy Deb. A simulated annealing-based multiobjective optimization algorithm: AMOSA. *IEEE Transactions on Evolutionary Computation*, 12(3):269–283, 2008 (cited on pages 27, 35).

[BSS+07] Michael Bauland, Thomas Schneider, Henning Schnoor, Ilka Schnoor, and Heribert Vollmer. The complexity of generalized satisfiability for linear temporal logic. In Helmut Seidl, editor, *Foundations of Software Science and Computational Structures*, pages 48–62, Berlin, Heidelberg. Springer Berlin Heidelberg, 2007 (cited on pages 6, 57, 110).

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010 (cited on page 17).

[CA17] Davide G. Cavezza and Dalal Alrajeh. Interpolation-based gr(1) assumptions refinement. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 281–297, Berlin, Heidelberg. Springer Berlin Heidelberg, 2017 (cited on page 13).

[CBR$^+$01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19:7–34, January 2001 (cited on pages 78, 114).

[CCG$^+$02] Alessandro Cimatti, E. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, M. Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: an opensource tool for symbolic model checking. *14th International Conference, CAV, Copenhagen, Denmark*, January 2002 (cited on pages 4, 57, 78, 92, 107, 114, 120).

[CDB$^+$23] 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. Springer Nature Switzerland, 2023 (cited on pages 23, 64, 82, 92, iii).

[CDC$^+$24] Luiz Carvalho, Renzo Degiovanni, Maxime Cordy, Nazareno Aguirre, Yves Le Traon, and Mike Papadakis. Specbcfuzz: fuzzing ltl solvers with boundary conditions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, Lisbon, Portugal. Association for Computing Machinery, 2024 (cited on pages 18, 76, 108, 109, 111, 115, 116, 118–121, 125, iii).

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg. Springer Berlin Heidelberg, 1982 (cited on page 4).

[CGH97] E. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. *Another look at ltl model checking*. In 1997 (cited on pages 78, 114).

[CGP$^+$02a] 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, pages 49–56, Berlin, Heidelberg. Springer-Verlag, 2002 (cited on pages 78, 114).

[CGP$^+$02b] 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. Springer Berlin Heidelberg, 2002 (cited on pages 92, 120).

[CGS+13] 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. Springer Berlin Heidelberg, 2013 (cited on page 91).

[CHJ08] 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. Springer Berlin Heidelberg, 2008 (cited on page 13).

[CHV+18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking.* Springer Publishing Company, Incorporated, 1st edition, 2018 (cited on pages 2, 60, 77, 78, 107, 109, 110).

[CR04] Carlos A. Coello Coello and Margarita Reyes Sierra. A study of the parallelization of a coevolutionary multi-objective evolutionary algorithm. In Raúl Monroy, Gustavo Arroyo-Figueroa, Luis Enrique Sucar, and Humberto Sossa, editors, *MICAI 2004: Advances in Artificial Intelligence*, pages 688–697, Berlin, Heidelberg. Springer Berlin Heidelberg, 2004 (cited on page 44).

[CTM+17] Alberto Camacho, Eleni Triantafillou, Christian Muise, Jorge Baier, and Sheila McIlraith. Non-deterministic planning with temporally extended goals: ltl over finite and infinite traces. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), February 2017 (cited on pages 4, 6, 74, 76).

[CvL15] Antoine Cailliau and Axel van Lamsweerde. Handling knowledge uncertainty in risk-based requirements engineering. In *IEEE International Requirements Engineering Conference, RE 2015, Ottawa, ON, Canada, August 24-28, 2015*, pages 106–115, 2015 (cited on pages 23, 57).

[CWB15] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741, 2015 (cited on page 80).

[DAA+14] Renzo Degiovanni, Dalal Alrajeh, Nazareno Aguirre, and Sebastián Uchitel. Automated goal operationalisation based on interpolation and sat solving. In *International Conference on Software Engineering*, pages 129–139, 2014 (cited on pages 12, 23).

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering*, pages 411–420, 1999 (cited on pages 4, 57, 107).

[DBP$^+$13] 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 (cited on page 14).

[DCA$^+$18] Renzo Degiovanni, Pablo F. Castro, Marcelo Arroyo, Marcelo Ruiz, Nazareno Aguirre, and Marcelo F. Frias. Goal-conflict likelihood assessment based on model counting. In *International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1125–1135, 2018 (cited on pages 13, 23, 30, 41, 43, 57, 64, 92, 120).

[DDM$^+$08] M. De Wulf, L. Doyen, N. Maquet, and J. -F. Raskin. Antichains: alternative algorithms for ltl satisfiability and model-checking. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 63–77, Berlin, Heidelberg. Springer Berlin Heidelberg, 2008 (cited on page 117).

[Dic45] Lee R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945 (cited on pages 94, 98).

[DJ14] 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 (cited on pages 27, 35, 40, 84, 101, 124).

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008 (cited on pages 6, 74, 76).

[dMB08] 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. Springer Berlin Heidelberg, 2008 (cited on pages 17, 91, 120).

[DMR$^+$18] Renzo Degiovanni, Facundo Molina, Germán Regis, and Nazareno Aguirre. A genetic algorithm for goal-conflict identification. In *International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 520–531, 2018 (cited on pages 13, 30, 40–43, 64, 80, 92, 93, 120).

[DRA$^+$16] Renzo Degiovanni, Nicolás Ricci, Dalal Alrajeh, Pablo F. Castro, and Nazareno Aguirre. Goal-conflict detection based on temporal satisfiability checking. In *International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 507–518, 2016 (cited on pages 12, 23, 30, 41, 64, 92, 120).

[DvLF93] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. In *Science of Computer Programming*, pages 3–50, 1993 (cited on pages 12, 23, 57, 80, 107).

[EBM$^+$12] Neil A. Ernst, Alexander Borgida, John Mylopoulos, and Ivan J. Jureta. Agile requirements evolution via paraconsistent reasoning. In *Proc. of the 24th Intl. Conf. on Advanced Information Systems Engineering*, pages 382–397, 2012 (cited on page 12).

[EC82] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.*, 2(3):241–266, 1982 (cited on page 13).

[EKS20] Javier Esparza, Jan Křetínský, and Salomon Sickert. A unified translation of linear temporal logic to -automata. *J. ACM*, 67(6), October 2020 (cited on pages 62, 64, 70).

[Eme85] E. Allen Emerson. Automata, tableaux and temporal logics (extended abstract). In *Proceedings of the Conference on Logic of Programs*, pages 79–88, Berlin, Heidelberg. Springer-Verlag, 1985 (cited on page 99).

[ESH14] Christian Ellen, Sven Sieverding, and Hardi Hungar. Detecting consistencies and inconsistencies of pattern-based functional requirements. In *Proc. of the 19th Intl. Conf. on Formal Methods for Industrial Critical Systems*, pages 155–169, 2014 (cited on page 12).

[FHI$^+$21] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Sat competition 2020. *Artificial Intelligence*, 301:103572, 2021 (cited on page 17).

[FJR09] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. An antichain algorithm for ltl realizability. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 263–277, Berlin, Heidelberg. Springer Berlin Heidelberg, 2009 (cited on page 117).

[FL79] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979 (cited on pages 92, 120).

[FME$^+$20] 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. USENIX Association, 2020 (cited on pages 79, 110).

[FT14] 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 (cited on pages 26, 86).

[GDP$^+$95] Rob Gerth, Den Dolech, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. *Proceedings of the 6th Symposium on Logic in Computer Science*, 15, December 1995 (cited on page 61).

[GGM$^+$21a] 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*, 8:1–8:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021 (cited on pages 6, 78, 110).

[GGM$^+$21b] 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)*, volume 206 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 8:1–8:17, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021 (cited on pages 76, 78, 91, 114, 119, 120).

[GGM21] 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, 2021 (cited on pages 78, 91, 119).

[GKC13] 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. Springer Berlin Heidelberg, 2013 (cited on page 17).

[GMR24] Ariel Gorenstein, Shahar Maoz, and Jan Oliver Ringert. Kind controllers and fast heuristics for non-well-separated gr(1) specifications. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE '24, Lisbon, Portugal. Association for Computing Machinery, 2024 (cited on pages 52, 57).

[GNR$^+$21] Jean-Raphaël Gaglione, Daniel Neider, Rajarshi Roy, Ufuk Topcu, and Zhe Xu. Learning linear temporal properties from noisy data: a maxsat-based approach. In Zhe Hou and Vijay Ganesh, editors, *Automated Technology for Verification and Analysis*, pages 74–90, Cham. Springer International Publishing, 2021 (cited on pages 5, 58, 59, 63, 65).

[GRT18] Alberto Griggio, Marco Roveri, and Stefano Tonetta. Certifying proofs for ltl model checking. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9, 2018 (cited on pages 96, 97).

[Gup93] Aarti Gupta. *Formal hardware verification methods: a survey.* In *Computer-Aided Verification: A Special Issue of Formal Methods In System Design on Computer-Aided Verification.* Robert Kurshan, editor. Springer US, Boston, MA, 1993, pages 5–92 (cited on pages 4, 6, 57, 74, 76).

[HAB$^+$05] Constance L. Heitmeyer, Myla Archer, Ramesh Bharadwaj, and Ralph D. Jeffords. Tools for constructing requirements specifications: the scr toolset at the age of ten. *Comput. Syst. Sci. Eng.*, 20(1), 2005 (cited on page 12).

[HGM$^+$21] 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, pages 230–243, Virtual, Denmark. Association for Computing Machinery, 2021 (cited on page 80).

[HHT02] J.H. Hausmann, R. Heckel, and G. Taentzer. Detection of conflicting functional requirements in a use case-driven approach. In *ICSE*, pages 105–115, 2002 (cited on page 12).

[HJ01] Mark Harman and Bryan F Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001 (cited on pages 27, 83).

[HJS09] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *JSAT*, 6:245–262, June 2009 (cited on pages 112, 113).

[HKP05] David Harel, Hillel Kugler, and Amir Pnueli. Synthesis revisited: generating statechart models from scenario-based requirements. In *Formal Methods in Software and Systems Modeling: Essays Dedicated to Hartmut Ehrig on the Occasion of His 60th Birthday*, pages 309–324, 2005 (cited on page 12).

[HLH97] Bernardo Huberman, Rajan Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science (New York, N.Y.)*, 275:51–4, February 1997 (cited on page 112).

[HMZ12] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: trends, techniques and applications. *ACM Comput. Surv.*, 45(1), December 2012 (cited on pages 27, 83).

[HN98] Anthony Hunter and Bashar Nuseibeh. Managing inconsistent specifications: reasoning, analysis, and action. *ACM TOSEM*, 7(4):335–367, 1998 (cited on page 12).

[Hol92] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992 (cited on pages 28, 35, 40).

[Hol97] G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997 (cited on pages 4, 57, 107).

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and models of concurrent systems*, chapter On the development of reactive systems, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985 (cited on pages 4, 52, 107).

[HS02] Ullrich Hustadt and Renate A. Schmidt. Scientific Benchmarking with Temporal Logic Decision Procedures. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning*, pages 533–546. Morgan Kaufmann, 2002 (cited on page 117).

[IEE98] IEEE. *IEEE Recommended Practice for Software Requirements Specifications.* IEEE Std 830-1998, 1998 (cited on page 23).

[Jac06] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis.* MIT Press, 2006, pages I–XVI, 1–350 (cited on pages 5, 13).

[Jac12] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2012 (cited on page 12).

[JXZ$^+$18] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 298–309, Amsterdam, Netherlands. Association for Computing Machinery, 2018 (cited on page 37).

[Kam09] M. Kamalrudin. Automated software tool support for checking the inconsistency of requirements. In *ASE*, pages 693–697, 2009 (cited on page 12).

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983 (cited on page 27).

[KHG11] Massila Kamalrudin, John Hosking, and John Grundy. Improving requirements quality using essential use case interaction patterns. In *ICSE*, pages 531–540, 2011 (cited on page 12).

[KL21] Hong Jin Kang and David Lo. Adversarial specification mining. *ACM Trans. Softw. Eng. Methodol.*, 30(2), January 2021 (cited on page 14).

[KMS$^+$83] 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 (cited on page 29).

[KMS18] 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 (cited on pages 44, 64, 70, 93, 101, 121, 124).

[Koy92] Ron Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*, volume 651 of *Lecture Notes in Computer Science.* Springer, 1992 (cited on page 12).

[KSO23] 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 (cited on page 17).

[KV99] Orna Kupferman and Moshe Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19, September 1999 (cited on page 57).

[KVW00] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, March 2000 (cited on page 79).

[KW52] William H Kruskal and W Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952 (cited on page 44).

[Lam00] Axel van Lamsweerde. Formal specification: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 147–159, Limerick, Ireland. Association for Computing Machinery, 2000 (cited on page 3).

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., USA, 2002 (cited on pages 4, 6, 57, 78, 107, 110).

[Let01] Emmanuel Letier. *Reasoning about Agents in Goal-Oriented Requirements Engineering.* PhD thesis, Université catholique de Louvain, 2001 (cited on pages 31, 32).

[Let02] Emanuel Letier. Goal-oriented elaboration of requirements for a safety injection control system. Technical report, Université catholique de Louvain, 2002 (cited on page 23).

[Lev66] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, February 1966 (cited on page 85).

[LFD$^+$19] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. Formal specification and verification of autonomous robotic systems: a survey. *ACM Computing Surveys*, 52(5), September 2019 (cited on page 12).

[LM19] Maciej Laszczyk and Paweł B. Myszkowski. Survey of quality measures for multi-objective optimization: construction of complementary set of multi-objective quality measures. *Swarm and Evolutionary Computation*, 48:109–133, 2019 (cited on page 44).

[LMS20] Michael Luttenberger, Philipp J. Meyer, and Salomon Sickert. Practical synthesis of reactive systems from LTL specifications via parity games. *Acta Informatica*, 57(1-2):3–36, 2020 (cited on page 57).

[LNA$^+$03] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim Mcfadden, and Yoav Shoham. A portfolio approach to algorithm selection. *IJCAI International Joint Conference on Artificial Intelligence*, May 2003 (cited on pages 112, 113).

[LNS02] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Learning the empirical hardness of optimization problems: the case of combinatorial auctions. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, pages 556–572, Berlin, Heidelberg. Springer Berlin Heidelberg, 2002 (cited on pages 112, 113).

[LPB15] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General ltl specification mining (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 81–92, 2015 (cited on pages 5, 58, 59, 62, 65).

[LPZ$^+$13] 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 (cited on pages 18, 44, 118).

[LS18] Caroline Lemieux and Koushik Sen. Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, pages 475–485, Montpellier, France. Association for Computing Machinery, 2018 (cited on page 80).

[LWS$^+$21] 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 (cited on pages 13, 23, 30, 41, 43, 120).

[LY19] Miqing Li and Xin Yao. Quality evaluation of solution sets in multiobjective optimisation: a survey. *ACM Computing Surveys*, 52(2), 2019 (cited on pages 24, 37, 44).

[LYP$^+$14] Jianwen Li, Yinbo Yao, Geguang Pu, Lijun Zhang, and Jifeng He. Aalta: an ltl satisfiability checker over infinite/finite traces. In FSE 2014, pages 731–734, Hong Kong, China. Association for Computing Machinery, 2014 (cited on page 79).

[LZP$^+$13] 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 (cited on pages 91, 99).

[LZP$^+$15] 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. Springer International Publishing, 2015 (cited on pages 91, 99, 115, 119).

[Mar05] 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. Springer Berlin Heidelberg, 2005 (cited on pages 92, 120).

[MB03] Jean-François Monin and Michael BSc. *Understanding formal methods*. In January 2003, pages 125–148 (cited on page 2).

[MCW$^+$20] 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, pages 701–712, Virtual Event, USA. Association for Computing Machinery, 2020 (cited on pages 17, 19).

[MDL$^+$22] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. Linear-time temporal logic guided greybox fuzzing. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1343–1355, 2022 (cited on page 80).

[MFS90] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990 (cited on pages 79, 110).

[MHH$^+$21] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: a survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021 (cited on page 111).

[MKT$^+$15] P.K. Murukannaiah, A.K. Kalia, P.R. Telangy, and M.P. Singh. Resolving goal conflicts via argumentation-based analysis of competing hypotheses. In *Proc. 23rd IEEE Int. Requirements Engineering Conf.* Pages 156–165, 2015 (cited on page 12).

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992 (cited on pages 4, 24, 25, 57, 74, 81).

[MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety.* Springer-Verlag New York, Inc., New York, NY, USA, 1995 (cited on page 12).

[MR16] Shahar Maoz and Jan Oliver Ringert. On well-separation of GR(1) specifications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 362–372, 2016. DOI: `10.1145/2950290.2950300`. URL: `https://doi.org/10.1145/2950290.2950300` (cited on page 14).

[MR21] Shahar Maoz and Jan Oliver Ringert. Spectra: a specification language for reactive systems. *Softw. Syst. Model.*, 20(5):1553–1586, October 2021 (cited on page 57).

[MRS19] Shahar Maoz, Jan Oliver Ringert, and Rafi Shalom. Symbolic repairs for GR(1) specifications. In *International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 1016–1026, 2019 (cited on page 13).

[MW47] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947 (cited on pages 44, 101).

[MW84] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, 1984 (cited on page 13).

[NDV15] 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, pages 1093–1100, Madrid, Spain. Association for Computing Machinery, 2015 (cited on pages 44, 93, 101, 124).

[Nis99] N. Nissanke. *Formal Specification: Techniques and Applications.* Springer London, 1999 (cited on page 3).

[NP21] 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 (cited on page 80).

[NQR+13] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: program repair via semantic analysis. In *International Conference on Software Engineering*, pages 772–781, 2013 (cited on page 37).

[NSG+18] Chi Mai Nguyen, Roberto Sebastiani, Paolo Giorgini, and John Mylopoulos. Multi-objective reasoning with constrained goal models. *Requirements Engineering*, 23(2):189–225, 2018 (cited on pages 23, 57, 80).

[NVL+13] Tuong Huan Nguyen, Bao Quoc Vo, Markus Lumpe, and John Grundy. KBRE: a framework for knowledge-based requirements engineering. *Software Quality Journal*, 22(1):87–119, 2013 (cited on page 12).

[OL82] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, July 1982 (cited on pages 4, 6, 57, 107).

[PLS+19] 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, pages 329–340, Beijing, China. Association for Computing Machinery, 2019 (cited on pages 79, 110, 111).

[PLS19] 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, pages 398–401, Beijing, China. Association for Computing Machinery, 2019 (cited on pages 79, 111).

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977 (cited on pages 4, 57, 107).

[PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 364–380, 2006 (cited on pages 4, 57, 107).

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, Austin, Texas, USA. ACM, 1989 (cited on page 13).

[RCS+22] M. Roggenbach, A. Cerone, B.H. Schlingloff, G. Schneider, and S.A. Shaikh. *Formal Methods for Software Engineering: Languages, Methods, Application Domains*. Texts in Theoretical Computer Science. An EATCS Series. Springer International Publishing, 2022 (cited on page 2).

[Rey16a] Mark Reynolds. A new rule for ltl tableaux. In *International Symposium on Games, Automata, Logics and Formal Verification*, 2016 (cited on pages 91, 114, 115, 119, 120).

[Rey16b] Mark Reynolds. A new rule for ltl tableaux. *Electronic Proceedings in Theoretical Computer Science*, 226:287–301, September 2016 (cited on page 78).

[RGB+23] Rajarshi Roy, Jean-Raphaël Gaglione, Nasim Baharisangari, Daniel Neider, Zhe Xu, and Ufuk Topcu. Learning interpretable temporal properties from positive examples only. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(5):6507–6515, June 2023 (cited on pages 5, 15).

[Ric76] John R. Rice. The algorithm selection problem. In Morris Rubinoff and Marshall C. Yovits, editors. Volume 15, Advances in Computers, pages 65–118. Elsevier, 1976 (cited on page 112).

[RJK+17] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: application-aware evolutionary fuzzing. In *Network and Distributed System Security Symposium*, 2017 (cited on pages 80, 111).

[RLS04] Kenneth J. Rothman, Stephan Lanes, and Susan T. Sacks. The reporting odds ratio and its advantages over the proportional reporting ratio. *Pharmacoepidemiology and Drug Safety*, 13(8):519–523, 2004 (cited on page 121).

[Roz11] Kristin Y. Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163–203, 2011 (cited on pages 4, 57, 61, 107).

[RRF+22] Ritam Raha, Rajarshi Roy, Nathanaël Fijalkow, and Daniel Neider. Scalable anytime algorithms for learning fragments of linear temporal logic. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 263–280, Cham. Springer International Publishing, 2022 (cited on page 15).

[RV07] Kristin Y. Rozier and Moshe Y. Vardi. Ltl satisfiability checking. In Dragan Bošnački and Stefan Edelkamp, editors, *Model Checking Software*, pages 149–167, Berlin, Heidelberg. Springer Berlin Heidelberg, 2007 (cited on page 61).

[RV10] Kristin Rozier and Moshe Vardi. Ltl satisfiability checking. *STTT*, 12:123–137, January 2010 (cited on pages 117, 118).

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, July 1985 (cited on pages 6, 57, 110).

[Sch98a] Stefan Schwendimann. A new one-pass tableau calculus for pltl. In *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, TABLEAUX '98, pages 277–292, Berlin, Heidelberg. Springer-Verlag, 1998 (cited on pages 78, 114).

[Sch98b] 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. Springer Berlin Heidelberg, 1998 (cited on page 91).

[SD11] 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, pages 397–413, Taipei, Taiwan. Springer-Verlag, 2011 (cited on pages 6, 17, 18, 74, 78, 108, 110, 113–115, 117–121, 123, 125).

[SF97] George Spanoudakis and Anthony Finkelstein. Reconciling requirements: a method for managing interference, inconsistency and conflict. *Annals of Software Engineering*, 3(1):433–457, 1997 (cited on page 12).

[Sis94] A. Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6:495–511, 1994 (cited on page 57).

[SM07] Horst Samulowitz and Roland Memisevic. Learning to solve qbf. In volume 1, pages 255–260, January 2007 (cited on page 113).

[SNC09] 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. Springer Berlin Heidelberg, 2009 (cited on page 91).

[SOR48] T. SORENSEN. A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on danish commons. *Biologiske Skrifter*, 5:1–34, 1948 (cited on pages 94, 98).

[SS24] Dominik Schreiber and Peter Sanders. Mallobsat: scalable sat solving by clause sharing. *Journal of Artificial Intelligence Research*, 80:1437–1495, August 2024 (cited on page 112).

[SWZ$^+$17] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. Automated test generation and mutation testing for alloy. In *IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 264–275, 2017 (cited on page 13).

[SYW$^+$23] 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 (cited on pages 7, 17, 108).

[SZ22] 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, pages 583–594, Singapore, Singapore. Association for Computing Machinery, 2022 (cited on pages 79, 80, 110, 111).

[TI20] Ryoji Tanabe and Hisao Ishibuchi. An analysis of quality indicators using approximated optimal distributions in a 3-d objective space. *IEEE Transactions on Evolutionary Computation*, 24(5):853–867, 2020 (cited on page 44).

[Tre08] Jan Tretmans. *Model based testing with labelled transition systems.* In *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*. Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pages 1–38 (cited on page 3).

[Var96] Moshe Vardi. An automata-theoretic approach to linear temporal logic. *Logics for Concurrency*:238–266, 1996 (cited on page 79).

[VD00] 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 (cited on pages 44, 94, 100).

[vLam09] Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications.* Wiley, 2009 (cited on pages 4, 12, 23, 25, 29, 57, 64, 74, 80, 107).

[vLDL98a] Axel van Lamsweerde, Robert Darimont, and Emmanuel Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926, 1998 (cited on pages 12, 23, 29, 30, 41, 80).

[vLDL98b] Axel van Lamsweerde, Robert Darimont, and Emmanuel Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineerings*, 24(11):908–926, 1998 (cited on pages 64, 80, 92, 120).

[vLL00] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Trans. Softw. Eng.*, 26(10):978–1005, October 2000 (cited on page 23).

[vLL98] Axel van Lamsweerde and Emmanuel Letier. Integrating obstacles in goal-driven requirements engineering. In *International Conference on Software Engineering*, ICSE '98, pages 53–62, Kyoto, Japan. IEEE Computer Society, 1998 (cited on pages 29, 80).

[WCG$^+$13] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 511–522, Berlin, Germany. Association for Computing Machinery, 2013 (cited on page 80).

[WCW$^+$18] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. In *International Conference on Software Engineering*, ICSE '18, pages 1–11, Gothenburg, Sweden. Association for Computing Machinery, 2018 (cited on page 37).

[Wil45] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945 (cited on pages 44, 94, 121).

[Wol85] Pierre Wolper. The tableau method for temporal logic: an overview. *Logique et Analyse*, 28(110/111):119–136, 1985. (Visited on 05/17/2023) (cited on pages 92, 120).

[WSK19] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. Arepair: a repair framework for alloy. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 103–106, 2019 (cited on pages 5, 13).

[XHH+08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *J. Artif. Int. Res.*, 32(1):565–606, June 2008 (cited on page 112).

[XHH+12] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Evaluating component solver contributions to portfolio-based algorithm selectors. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 228–241, Berlin, Heidelberg. Springer Berlin Heidelberg, 2012 (cited on pages 107, 112).

[YHT+21] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Fuzzing smt solvers via two-dimensional input space exploration. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, pages 322–335, Virtual, Denmark. Association for Computing Machinery, 2021 (cited on page 111).

[Yu97] Eric S. K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *IEEE International Symposium on Requirements Engineering*, RE '97, pages 226–, Washington, DC, USA. IEEE Computer Society, 1997 (cited on page 12).

[ZS03] Bin Zhang and Sargur N Srihari. Properties of binary vector dissimilarity measures. In *Proc. JCIS Int'l Conf. Computer Vision, Pattern Recognition, and Image Processing*, volume 1, pages 1–4, 2003 (cited on page 94).

[ZTL+03] Eckart Zitzler, Lothar Thiele, Marco Laumanns, Carlos M. Fonseca, and Viviane Grunert da Fonseca. Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transactions on Evolutionary Computation*, 7:117–132, 2003 (cited on page 44).

[ZWC+22] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Comput. Surv.*, 54(11s), September 2022 (cited on pages 79, 110).