University of Brasília

Institute of Exact Sciences
Department of Computer Science

Diagnóstico de rastreio para Propriedades de Sinais Temporais

Gabriel Frutuoso Pereira Araújo

Brasília
2024

University of Brasília

Institute of Exact Sciences
Department of Computer Science

Trace-Diagnostic for Signal Temporal Properties: An Evolutionary Approach

Gabriel Frutuoso Pereira Araújo

Thesis presented in partial fulfillment of the requirements for the degree of Master of Science in Informatics

Advisor
Prof.a Dr.a Genaína Nunes Rodrigues

Brazil
2024

University of Brasília
Institute of Exact Sciences
Department of Computer Science

Trace-Diagnostic for Signal Temporal Properties: An Evolutionary Approach

Gabriel Frutuoso Pereira Araújo

Thesis presented in partial fulfillment of the requirements for the degree of Master of Science in Informatics

Prof.a Dr.a Genaína Nunes Rodrigues (Advisor)
CIC/UnB

Prof. Dr. Rodrigo Bonifácio De Almeida
CIC/UnB

Prof. Dr. Lars Grunske
Institut für Informatik/HUB

Prof. Dr. Rodrigo Bonifácio
Coordinator of the Graduate Program in Informatics

Brazil, Brasília, October 18 2024

Dedication

Dedico esse trabalho aos meus pais, Maria de Lourdes e Evaldo Araújo, e à minha irmã, Joana Patrícia.

Acknowledgements

I would like to express my gratitude to everyone who made this work possible. First and foremost, I extend my heartfelt thanks to my supervisor, Genaína Nunes, whose trust and guidance were fundamental throughout this journey. I am also grateful to Claudio Menghi for suggesting the idea for this work and providing valuable insights and discussions that enriched it significantly. A big thank you to Patrizio Pelliccione, whose sharp reasoning and pragmatic mindset helped steer this project toward achievable results. I appreciate my colleague and friend, Ricardo Diniz, whom I have known since our undergraduate days; his assistance in double-checking my decisions regarding implementation and research choices has been invaluable. Thank you for our discussions.

I also want to thank Federico Formica; his involvement in the experimentation phase, along with his numerous suggestions, has greatly enhanced the maturity of this work beyond what I could have achieved alone.

Lastly, I am grateful to my alma mater, Universidade de Brasília, whose support and resources were essential in helping me achieve these results.

"Man is the measure of all things"

Protagoras of Abdera

Abstract

Cyber-physical systems (CPS) such as satellites, self-driving cars, service robots, and IoTs are in our daily lives. These systems must satisfy requirements specifying their operation over time. During the development of such systems, designers and engineers must test whether the implementation meets its specifications. In addition, in the case of a violation, they need to identify and diagnose where the failure comes from. Understanding such violations is especially crucial in safety-critical systems.

This work presents a novel technique to diagnose any system using only its requirements and test traces. Leveraging techniques like trace-checking and genetic programming, we deliver an informative diagnosis. The diagnosis shows to engineers what changes are sufficient to satisfy the violation requirement. The user can also customize the approach to focus on specific information relevant to the user.

We evaluate our approach in two verticals: accuracy and efficiency. We evaluate the capability of our approach in delivering informative diagnoses and the time it takes to provide these diagnoses. Our approach shows that it can produce an informative output for most of our experiments in a reasonable time. The tool exceeded its time budget for the remaining experiments, not producing any diagnosis.

Keywords: Diagnostics, Trace checking, Run-time verification, Temporal properties, Cyber-physical systems, Signals

Resumo

Sistemas ciber-físicos (CPS), como satélites, carros autônomos, robôs de serviço e IoTs, estão presentes em nossas vidas diárias. Esses sistemas devem atender a requisitos que especificam seu funcionamento ao longo do tempo. Durante o desenvolvimento de tais sistemas, designers e engenheiros devem testar se a implementação atende às suas especificações. Além disso, em caso de violação, é necessário identificar e diagnosticar de onde vem a falha. Compreender tais violações é especialmente crucial em sistemas críticos de segurança.

Este trabalho apresenta uma técnica inovadora para diagnosticar qualquer sistema utilizando apenas seus requisitos e traços de teste. Aproveitando técnicas como verificação de traços e programação genética, fornecemos um diagnóstico informativo. O diagnóstico mostra aos engenheiros quais mudanças são suficientes para satisfazer o requisito violado. O usuário também pode personalizar a abordagem para focar em informações específicas relevantes para o seu contexto.

Nós avaliamos nossa abordagem em duas vertentes: acurácia e eficácia. Avaliamos a capacidade de nossa abordagem em fornecer diagnósticos informativos e o tempo necessário para gerar esses diagnósticos. Nossa abordagem mostrou que pode produzir um resultado informativo para a maioria de nossos experimentos em um tempo razoável. A ferramenta excedeu o limite de tempo em alguns experimentos, não produzindo diagnóstico nesses casos.

Palavras-chave: Diagnóstico, Verificação de rastro, Verificação em tempo-real, Propriedades temporais, Sistemas Ciber-físicos, Sinais

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Research Challenges . . . 3
1.3 Research Questions and Evaluation . . . 4
1.3.1 Research Question 1 . . . 4
1.3.2 Research Question 2 . . . 4
1.4 Research Contributions . . . 5
1.5 Document Roadmap . . . 5

2 Running Example **6**
2.1 The Example . . . 6
2.2 Hybrid Logic of Signals . . . 8
2.3 ThEodorE . . . 9

3 Diagnosis **11**
3.1 Search in Search-Based Trace-Diagnostic . . . 11
3.2 Search-based Trace-Diagnostic . . . 12
3.3 Search-based Trace Diagnostic for HLS . . . 15
3.3.1 Change-Driven Diagnosis . . . 16
3.3.2 Generator of Mutations . . . 16
3.3.3 Trace-Checker . . . 20
3.3.4 Diagnostic Generator . . . 20

4 Evaluation and Discussion **22**
4.1 Experiment Setting and Tool Configuration . . . 22
4.2 Accuracy - EQ1 . . . 26
4.3 Efficiency - EQ2 . . . 30
4.4 Discussion and Threats to Validity . . . 31

5 Related Work **35**

6 Conclusion **37**

References **39**

List of Figures

1.1 Autonomous forklift tipping and dropping its load. 2

2.1 Example of failure-revealing scenario. 7
2.2 A fragment of an execution trace for our case study. 7
2.3 Syntax of HLS. 9
2.4 Schematic representation of ThEodorE trace-checker from [1]. 10

3.1 The search-based trace-diagnostic framework. 12
3.2 Diagnosis generated for our automotive example 14
3.3 AST of an HLS requirement and its mutation obtained by applying the mutation operator **OP13** to the node with the blue background. 15
3.4 Example of application of the over operator: the requirement ϕ_3 is obtained from the requirement ϕ_1 by swapping the subtree with the root node with a red background with the corresponding subtree from the requirement ϕ_2. 18
3.5 Example of fitness calculation using the score function of the Smith-Waterman algorithm. 19
3.6 Entries considered by the learning algorithm. 21

4.1 Diagnostic and prediction for the experiment *exp1*. 28
4.2 Precision and recall of SBTD across the different experiments. Diamonds depict the average, red lines are the median, and pluses depict the outliers. 30

List of Tables

3.1 Mutation Operators: the table contains the original formula and the mutated formula 15
3.2 Configuration Parameters for our SBTD framework. 17

4.1 Requirements from our benchmark. 24
4.2 HLS formalization for the requirements from 4.1. 25
4.3 Values for the configuration parameters of `Diagnosis` from 3.2. 26
4.4 Mutation operators (Operators) and ranges for the value terms (Ranges) of each experiment (Exp). 27
4.5 Time required by our SBTD tool to extract the diagnosis. 31

List of Acronyms

AST Abstract Syntax Tree

AT Automatic Transmission System

CC Chasing Cars System

CPS Cyber-Physical System

DT Decision Tree

EQ Evaluation Question

GA Genetic Algorithm

GP Genetic Programming

HLS Hybrid Logic of Signals

LTL Linear Temporal Logic

MTL Metric Temporal Logic

SBTD Search-Based Trace-Diagnostic

SMT Satisfiability Modulo Theories

STL Signal Temporal Logic

OPx Mutation Operator x

Chapter 1

Introduction

1.1 Motivation

In 2021, while working at an autonomous forklift start-up, we encountered a critical issue during field testing. Our navigation module was sending incorrect goal positions to the planner during docking, causing the robot to replan several times. Although, the system could recover from the problem, we decided to fix it as it led to repeated attempts when picking up pallets, which were time-consuming and undesired for the customer.

We tested the bugfix in a simulated warehouse before deployment, but when applying the fix in the actual robot, a small oversight in a parameter configuration resulted in a costly mistake. The robot picked up the pallet slightly off-center, causing it to drop and damage the goods. This incident highlighted to me how expensive and risky it is to test robots in real environments, where even minor errors in setup or debugging can lead to significant losses. Figure 1.1 shows an image generated based on the incident photograph to maintain anonymity.

Cyber-Physical Systems (CPS) testing is an important task that is typically achieved through various methods, like unit, integration and hardware testing. However, testing CPSs like robots in real life is expensive because it requires significant time for setup and execution, poses safety risks and involves the problem of debugging multiple modules at once. Although testing itself does not guarantee correctness, it increases confidence in system reliability [2]. The earlier mentioned episode illustrates how CPSs, despite prior testing, can still fail due to overlooked factors. Such failures accentuate the need for techniques that can help engineers quickly diagnose and correct bugs, ultimately reducing the cost and risk associated with real-world CPS testing. This leads to a critical question: how can engineers better ensure that CPS requirements are met and failures are minimized?

Figure (1.1) Autonomous forklift tipping and dropping its load[1].

To address this question, engineers rely on various testing techniques to detect requirement violations in complex CPSs. Many testing techniques (e.g., [3, 4, 5, 6, 7, 8, 9, 10]), compared by existing competitions (e.g., [11]), rely on *trace-checking* where system behavior for a specific input is recorded and evaluated against a requirement. For example, `ThEodorE` [1] is a trace-checking tool that supports requirements expressed using the Hybrid Logic of Signals (HLS) [12], an expressive logic to capture CPS requirements. Trace-checking techniques typically consider a trace and a property representing a requirement and return a Boolean verdict: *True* if the trace satisfies the requirement, and *False* otherwise. If the trace satisfies the requirement, testing tools automatically generate new test cases searching for a test case that violates the requirement. In the opposite case, engineers need to inspect the trace to understand the cause of the violation.

In response to this, *trace-diagnostic* techniques have emerged to explain why a violation occurred. Current approaches either isolate parts of the traces to identify the violation [13, 14, 15, 16] or analyze the traces for common behaviors leading to the violation [17, 18, 19, 20]. For example, Boufaied et al. [19] applied the latter method to diagnose violations

[1]Image generated using DALL-E.

in signal-based temporal requirements. The approach requires a language-specific library of violation causes and diagnoses. Then, it uses its database to search for an explanation for the requirement violation. Nevertheless, significant limitations were identified in this method.

Signal-temporal languages, while useful for some CPS requirements, struggle to fully capture the *discrete behaviors* often seen in CPS. Moreover, their catalog of violation causes was incomplete and unable to explain many root causes in their dataset. A core difficulty is that CPS, which often involves interactions with unpredictable physical environments, cannot anticipate all possible interactions. As a result, it's nearly impossible to create a complete catalog of violation causes in advance. Complex CPS like satellites, self-driving cars, service robots, and IoT systems feature numerous interconnected modules, and in many cases, engineers are left with only the system trace and specification to work from, making full system modeling infeasible.

1.2 Research Challenges

Despite the progress in trace-diagnostic techniques, two major challenges limit their practical application:

- Challenge *C1*: Incomplete Knowledge of System Behavior:

  The inherent complexity of CPS is a challenge. These systems often involve various components interacting in dynamic and unpredictable environments. Hence, engineers naturally will have limited knowledge of all potential interactions and failure modes prior to execution. This challenge makes pre-defining a complete set of violations and diagnoses difficult, opening diagnostic gaps when unexpected behaviors arise.

- Challenge *C2*: Unpredictability of Emergent Behaviors:

  Even when a library of violation causes exists, CPS often exhibits emergent behaviors — unexpected interactions between components or the environment that were not foreseen during design or testing. These emergent behaviors can lead to requirement violations that do not fit neatly within predefined categories, meaning existing diagnostic libraries may fail to provide an explanation. This unpredictability makes static approaches insufficient for fully diagnosing complex real-world systems.

This work mitigates these challenges by proposing search-based trace-diagnostic (SBTD), a novel trace-diagnostic framework for CPS. Unlike existing techniques, SBTD uses an

evolutionary search approach to generate new candidate diagnoses. This automated generation enables the dynamic creation of new diagnoses and provides two benefits. First, it addresses challenge *C1* since it does not require as input a library of predefined violation causes and diagnoses. This relieves engineers from the time-consuming and error-prone definition of such a library, when it is unavailable. Second, it resolves the challenge *C2* by enabling the dynamic creation of new diagnoses, which increases the likelihood of finding a valid explanation for a given violation. We defined `Diagnosis`, which is an instance of SBTD that considers properties modeled using the Hybrid Logic of Signals (HLS) [12].

1.3 Research Questions and Evaluation

1.3.1 Research Question 1

Given the presented Research Challenges (*C1* and *C2*), we can derive the following Research Questions:

> **RQ1**: How do we provide a diagnosis of a violation of the CPS requirement when there is no complete knowledge of the cause?

This question aims to propose a method for diagnosing requirement violations in scenarios where a library of causes is incomplete or unavailable before testing. In such cases, the focus shifts toward developing strategies to identify violations even when prior information is limited or missing. To address this question, we propose an approach in which the user can diagnose *any* system by utilizing only a violated requirement and a trace from its execution without any previous knowledge of the system. The details of this solution will be presented in Chapter 3.

1.3.2 Research Question 2

Once we have a diagnosis, we can also evaluate the provided solution giving also another Research Question:

> **RQ2**: How trustworthy are the generated diagnoses?

This study evaluates the trustworthiness of the proposed diagnostic method, focusing on accuracy and applicability of our solution in real-world CPS applications. We assess the accuracy of the generated diagnoses through metrics such as precision and recall. While applicability is determined using a benchmark comprising several case studies, designed to stress test the system as well as explore edge cases. To achieve this, we compare our

results with those of domain experts using precision and recall, while also evaluating the time required to generate a diagnosis.

We conducted 34 experiments involving 17 trace-requirement combinations that resulted in property violations across *three* systems: two automotive and one robotics. The effectiveness of our solution was measured by its ability to produce accurate and informative diagnoses, providing actionable recommendations for system improvement. A comprehensive discussion of our evaluation is presented in Chapter 5, which summarizes that our framework demonstrated high accuracy in most experiments, although some limitations were noted due to performance constraints.

1.4 Research Contributions

To summarize, our contributions are as follows:

- Search-based trace-diagnostic (SBTD), a novel trace-diagnostic technique for CPS based on evolutionary search (Chapter 3);
- An SBTD framework that supports properties expressed using the HLS (Section 3.3);
- The implementation of our SBTD framework, namely `Diagnosis`, which is publicly available [21];
- An extensive empirical evaluation of our solution (Chapter 4).

1.5 Document Roadmap

Our thesis is organized as follows. Chapter 5 discusses related work. Chapter 2 presents our running example from the automotive domain. Chapter 3 introduces SBTD framework. Chapter 4 evaluates our contribution and in Section 4.4 we reflect on our findings. Chapter 6 presents our conclusions.

Chapter 2

Running Example

Building on the challenges and questions discussed earlier, this chapter introduces a running example that will be used throughout the thesis to illustrate our approach. By following this example, the reader will better understand the concepts and techniques developed in this work. Moreover, it will serve as a reference point for interpreting the results and understanding how our approach functions in practice. The scenario, requirements, and trace presented here will be revisited in later chapters to explain key concepts in context. In the following chapter, we will detail the approach, using this example to walk through how it arrives at the expected outcomes.

2.1 The Example

Unlike the forklift example introduced earlier, which dealt with specific issues in real-world testing, this example is designed to help explain our approach in a clear and focused scenario. While this specific example will guide the reader through the key concepts in the thesis, our approach is evaluated later using this and other different systems, demonstrating its flexibility across various configurations. Our running example involves a vehicle that must follow a trajectory while avoiding obstacles, as illustrated in Figure 2.1. The solid line represents the intended trajectory, while the dashed line shows the actual path taken by the vehicle.

Automotive engineers analyze if their system behaves correctly by considering a set of driving scenarios. To specify the vehicle's behavior and verify its adherence to the trajectory-following requirement, engineers express the system's requirements using the model described by the Equation (2.1).

$$\phi ::= \forall \tau_0 \in [0, \infty), \quad \begin{cases} \texttt{d_pos_x}(\tau_0) - \texttt{v_pos_x}(\tau_0) < 20\text{cm} \\ \texttt{d2obs}(\tau_0) > 50\text{cm} \end{cases} \tag{2.1}$$

This requirement ϕ specifies that, for every time instant τ_0 from the beginning (time 0) to the end (∞) of the simulation, the two following conditions should hold:

1. the difference (`d_pos_x` **`@t`** (τ_0) $-$ `v_pos_x` **`@t`** (τ_0)) between the desired position (`d_pos_x`) and the actual vehicle position (`v_pos_x`) in the x-axis at time τ_0 is lower than a threshold value (20cm), and

2. the Euclidean distance (`d2obs`) between the vehicle's border and the obstacle's border is greater than the threshold value (50cm).

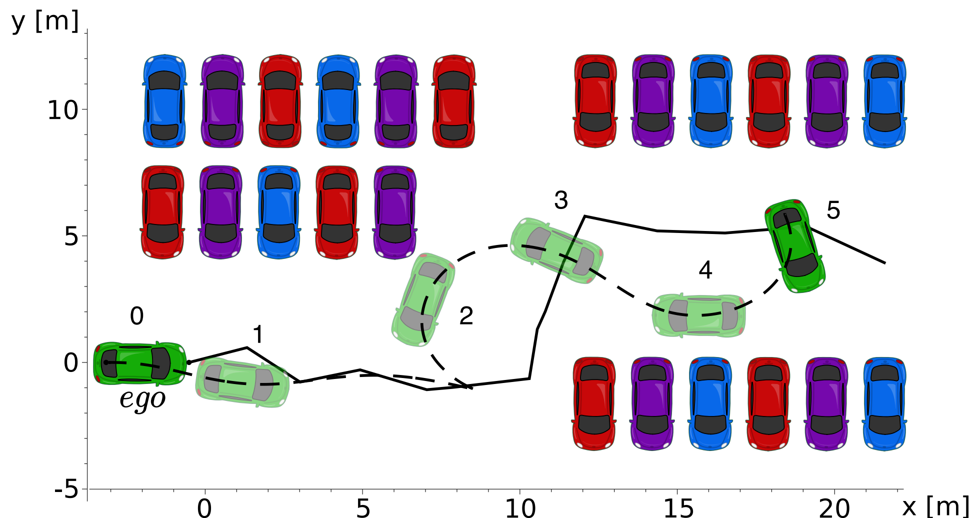


Figure (2.1) Example of failure-revealing scenario.

The scenario from Figure 2.1 represents a failure-revealing scenario in which the requirement ϕ is not satisfied: while following a desired trajectory (solid line), the course followed by the car (dashed line) causes the vehicle to reach a position ("2" labeled position) with a distance lower than 50cm from the obstacle.

`v_pos_x`	-0.15	-0.16	5.66	11.87	17.49	19.31
`d_pos_x`	-0.15	-0.16	7.86	14.56	19.09	19.31
`d2obs`	6.05	7.05	0.007	2.23	8.44	8.15
timestamp	0	1.0	5.0	11.0	12.5	15.0
position	0	1	2	3	4	5
			Record r_2			

Figure (2.2) A fragment of an execution trace for our case study.

Figure 2.2 reports a fragment of the execution trace for this driving scenario. Each position of the vehicle from Figure 2.1 is associated with a trace record that specifies the values assumed by the variables `v_pos_x`, `d_pos_x`, and `d2obs`, representing the actual position of the vehicle, the desired position of the vehicle, and the Euclidean distance

between vehicle's and the closest obstacle's border, at different simulation times. The values assumed by the variable timestamp represent the time instant of the different trace records. For example, the trace record r_2 specifies that at time instant 5.0s, the position of the vehicle is `v_pos_x` = 5.66m, the desired position is `d_pos_x` = 7.86m, and the distance to the closest object's border is `d2obs` = 0.007m.

When engineers inspect the execution trace, they need to identify the root causes of the failure. For example, for the scenario from Figure 2.1 and the corresponding trace in Figure 2.2, the vehicle fails to maintain the required safe distance of 50cm from the obstacle. At time instant 5.0s the distance is 7mm. Obtaining these explanations is usually challenging; requirements (e.g., expressed in HLS [12] or SB-TemPsy-DSL [19]) typically rely on many temporal operators and may have a complex structure. The goal of SBTD is to support engineers in automatically producing diagnostic information that can be useful for understanding the causes of the failure by relying on a search-based trace-diagnostic approach.

2.2 Hybrid Logic of Signals

The Hybrid Logic of Signals (HLS) was first introduced in [12] as a new and more expressive specification language compared to STL [22] and SB-TemPsy-DSL [23], tailored for specifying CPS requirements. HLS enables engineers to define properties that reference both time-stamps and indices in CPS traces, enabling easy specification of both cyber and physical component behaviors and their interactions.

Figure 2.3 presents the grammar of HLS from [12], where symbol "|" separates alternatives, TV is a set of timestamp variables, IV is a set of index variables, RV is a set of real-valued variables, and S is a set of signal variables.[1] An HLS *formula* (non-terminal `p`) is a relational expression over terms, a Boolean expression over formulae, or quantified formulae. Quantified formulae support quantification over timestamp variables ($\triangleright\, \tau$ **in** I_T $[\ldots]$), over index variables ($\triangleright\, \sigma$ **in** I_J $[\ldots]$), or over real-valued variables ($\triangleright\, \rho$ $[\ldots]$), where $\triangleright$ represents the existential (**exists**) or universal (**forall**) quantifier. A *term* (non-terminal `tm`) is a *time term*, an *index term*, or a *value term*. A *time term* (non-terminal `tt`) is a timestamp variable τ, a literal denoting a value t, the value returned by the operator **i2t** ("index to timestamp"), or an arithmetic expression over these entities. An *index term* (non-terminal `it`) is an index variable σ, a literal denoting a value j, the value returned by the operator **t2i** ("time to index"), or an arithmetic expression over these entities. A *value term* (non-terminal `vt`) is a real-valued variable ρ, a literal denot-

[1]We slightly revisited the presentation of the grammar to include derived operators (e.g., **forall**, **and**).

ing a value x, the value of a signal returned by the operators **@i** ("at index") and **@t** ("at timestamp"), or an arithmetic expression over these entities.

Formula	$\texttt{p} ::= \texttt{tm}_1 \oplus \texttt{tm}_2 \mid \textbf{not}\ \texttt{p} \mid \texttt{p}_1 \ominus \texttt{p}_2$ $\mid \triangleright\ \tau\ \textbf{in}\ I_T\ \textbf{such that}\ \texttt{p}$ $\mid \triangleright\ \sigma\ \textbf{in}\ I_J\ \textbf{such that}\ \texttt{p}$ $\mid \triangleright\ \rho\ \textbf{such that}\ \texttt{p}$
Term	$\texttt{tm} ::= \texttt{tt} \mid \texttt{vt} \mid \texttt{it}$
Time Term	$\texttt{tt} ::= \tau \mid t \mid \textbf{i2t}(\texttt{it}) \mid \texttt{tt}_1 \odot \texttt{tt}_2$
Index Term	$\texttt{it} ::= \sigma \mid j \mid \textbf{t2i}(\texttt{tt}) \mid \texttt{it}_1 \odot \texttt{it}_2$
Value Term	$\texttt{vt} ::= \rho \mid x \mid (s\ \textbf{@i}\ \texttt{it}) \mid (s\ \textbf{@t}\ \texttt{tt}) \mid \texttt{vt}_1 \odot \texttt{vt}_2$

$t, x \in \mathbb{R}, j \in \mathbb{N}^+, I_T \subseteq \mathbb{R}, I_J \subseteq \mathbb{N}^+, \tau \in TV, \sigma \in SV, \rho \in RV, s \in S$
$\odot \in \{+, -, *, /\}$;
$\oplus \in \{>, <, \leq, \geq, =, \neq\}$
$\ominus \in \{\textbf{or}, \textbf{and}, \textbf{implies}\}$
$\triangleright \in \{\textbf{exists}, \textbf{forall}\}$.

Figure (2.3) Syntax of HLS.

Therefore, using HLS, engineers specify the same requirement ϕ of the vehicle as:

$$\phi ::= \textbf{forall}\ \tau_0\ \textbf{in}\ [0, \infty)\ \textbf{such that}$$
$$(\texttt{d_pos_x}\ \textbf{@t}\ (\tau_0) - \texttt{v_pos_x}\ \textbf{@t}\ (\tau_0)) < \boxed{20}\,\text{cm}\ \boxed{\textbf{and}}\ \texttt{d2obs}\ \textbf{@t}\ (\tau_0) > \boxed{50}\,\text{cm}$$

The semantics of the background colored boxes will be defined in Chapter 3 and Section 3.3. We will use $\phi($ **and** $)$, $\phi($ 20 $)$, $\phi($ 50 $)$ to respectively indicate the operator **and** and the values 20 and 50 contained within the yellow, red, and blue colored background boxes of the requirement ϕ.

2.3 ThEodorE

`ThEodorE` was introduced in [1] as a tool for verifying traces based on properties defined in HLS specifications. It simplifies trace verification by translating the problem into a satisfiability task, which can be solved using a Satisfiability Modulo Theories (SMT) solver.

As an Eclipse plugin, `ThEodorE` accepts both a system specification and an execution trace. It then generates a Python script that leverages an SMT engine to check if the trace conforms to the given specification.

`ThEodorE` offers users an intuitive graphical interface within Eclipse. This includes a syntax checker for writing requirements and an automated process for verifying whether

those requirements hold true for execution traces of a Cyber-Physical System (CPS). Figure 2.4 shows the schematic representation of the ThEodorE trace-checker and its software components. The tool is available in a public repository in [24].

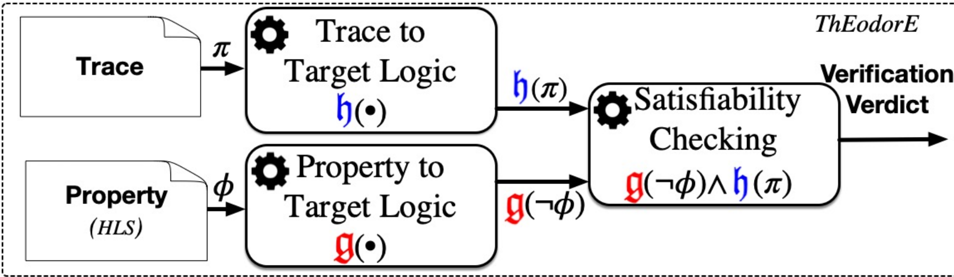


Figure (2.4) Schematic representation of ThEodorE trace-checker from [1].

Chapter 3

Diagnosis

In the previous chapter, we introduced our motivating example, outlining the inputs, outputs, and rationale behind our approach. While demonstrating how to interpret the results through a real-world case study. This established the need for effective diagnostics in complex CPS systems.

In this chapter, we focus on the detailed workings of our approach and its three key modules. We show their implementation and the reasoning behind the implementation choices. Throughout, we refer to the motivating example introduced earlier to clarify core concepts and illustrate how our approach operates in practice. The next chapter is dedicated to evaluating the performance and flexibility of our approach across various CPSs.

3.1 Search in Search-Based Trace-Diagnostic

Cyber-physical systems (CPS) are inherently complex due to dynamic interactions between components and their environment. This complexity extends to system requirements, which often include temporal operators and intricate structures. Diagnosing requirement violations in such systems presents significant challenges, particularly in determining how individual terms contribute to these violations. Unlike traditional methods relying on formal models, our approach is entirely model-independent, focusing solely on requirement terms, making it applicable even when constructing formal models is impractical.

Genetic programming (GP) [25] offers a natural solution to explore large search spaces effectively. Using crossover and mutation, GP generates diverse requirements variations, enabling a wider exploration of term interactions. Our model-independent approach derives insights directly from the structure of requirements without relying on formal models. A reasonably defined fitness function [26] is sufficient to guide the GP process effectively,

although experimenting with different functions may enhance performance by avoiding local optima and accelerating convergence.

One key challenge is comparing requirements involving a mix of logical, mathematical, and comparison operators, along with numerical values and time intervals. The expressiveness of these requirements complicates measuring the "distance" between them, necessitating further research. To address this complexity, we simplify by measuring the similarity between two properties rather than directly assessing intricate operator interactions, balancing implementation feasibility with effective exploration.

Gaaloul et al. [27] similarly applied GP to generate environment assumptions through model checking. However, our approach differs by being entirely model-agnostic, evolving requirement variations to uncover significant patterns associated with violations, thereby extending applicability to a wider range of systems without model dependencies.

By expanding the search space through tailored mutation and crossover operations, GP reveals patterns and relationships among terms that contribute to requirement violations. These insights are systematically compiled into a reusable catalog of violation causes, facilitating future diagnostics.

In conclusion, GP not only aids in diagnosing requirement violations but also provides a systematic, model-independent framework for analyzing and categorizing the impact of individual terms on system behavior. This model-free approach is particularly well suited for diverse CPS scenarios where formal models are unavailable, thereby enhancing its utility across various complex systems.

3.2 Search-based Trace-Diagnostic

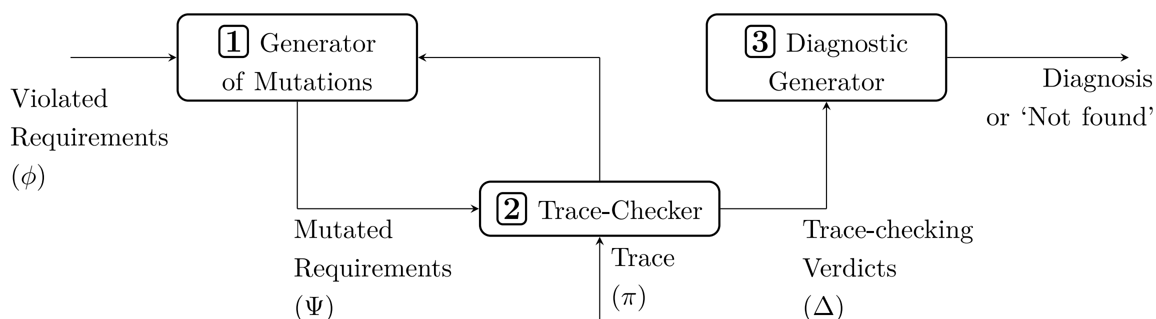


Figure (3.1) The search-based trace-diagnostic framework.

Figure 3.1 presents an overview of SBTD. The input of SBTD is a trace-requirement combination ($\langle \pi, \phi \rangle$) made by a requirement formalized as a property (ϕ) unsatisfied over the trace (π) and a time budget (b). SBTD either successfully returns a diagnosis d or

informs the user that it could not find a diagnosis within the available budget. SBTD works in three steps:

1. The *Generator of Mutations* step generates a set Ψ of candidate mutated requirements from a (set of) requirement(s). Our SBTD framework generates mutated requirements with high similarity with the original requirement ϕ which can more likely be informative in explaining the cause of the violation. For example, given the requirement formalized as ϕ of our running example from Chapter 2, the generator synthesizes the following mutated requirement ϕ' by changing the value $\phi(\,50\,)$ reported within the blue colored background box (50cm) into 45cm.

$$\phi' ::= \textbf{forall}\ \tau_0\ \textbf{in}\ [0,\infty)\ \textbf{such that}$$
$$(\texttt{d_pos_x}\ \texttt{@t}\ (\tau_0) - \texttt{v_pos_x}\ \texttt{@t}\ (\tau_0)) < 20\,\text{cm}\ \textbf{and}\ \texttt{d2obs}\ \texttt{@t}\ (\tau_0) > 45\,\text{cm}$$

2. The *Trace-Checker* step receives a set of mutated requirements Ψ and checks whether each mutated requirement is satisfied or violated by the trace π rendering a set of pairs each associating a trace-requirement combination with a Boolean value indicating whether the requirement is satisfied or violated over the corresponding trace. Considering our running example, when the trace-checker evaluates the mutated requirement ϕ', it detects that the trace π satisfies the requirement ϕ' and produces the pair $\{\langle\pi,\phi'\rangle, True\}$. The *Trace Checker* component produces a set Δ of pairs $\{\langle\pi,\phi'\rangle, \upsilon\}$ made by the trace π, the mutated requirement ϕ', and the corresponding trace checking verdict υ. However, to run the *Diagnostic Generator* step, it is necessary to have at least a certain number of satisfied and violated requirements within the set Δ, such that the *Diagnostic Generator* can produce an informative diagnosis. Therefore, the *Generator of Mutations* and the *Trace-Checker* are executed iteratively and the set Δ is augmented with the newly generated pairs until (at least) a certain number of satisfied and violated requirements are present.

3. The *Diagnostic Generator* step analyzes the requirement ϕ and the pairs containing the trace-checking verdicts of the mutated requirements (e.g., $\{\langle\pi,\phi'\rangle, True\}$) to produce a diagnosis. If it can not produce an informative diagnosis, it starts another iteration by running step 1 and by considering a new set of the mutated requirements. Otherwise, it returns the diagnosis to the user.

The algorithm stops by either outputting the informative diagnosis, if found within the time budget (b), or by prompting a message indicating that SBTD could not produce a diagnosis within the time budget.

To illustrate our methodology, Figure 3.2 presents a decision tree (DT) as the diagnosis of our SBTD for the running example.

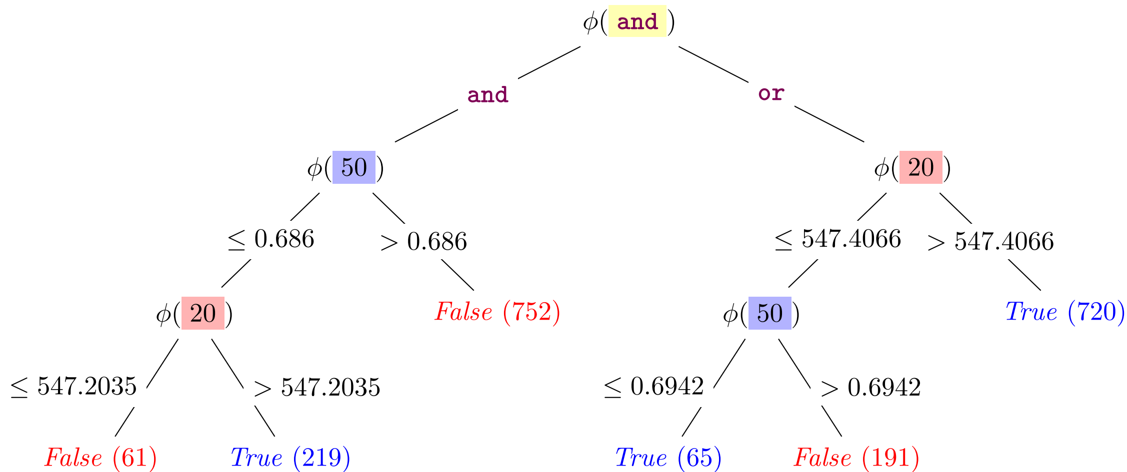


Figure (3.2) Diagnosis generated for our automotive example.[1]

The diagnosis highlights which sets of changes for $\phi($ **and** $)$, $\phi($ 50 $)$, $\phi($ 20 $)$ can make the formula satisfied. For example, for the considered trace, to make the requirement satisfied, the developer can maintain the **and** logical operator for $\phi($ **and** $)$, set a threshold value $\phi($ 20 $)$ for the difference between the desired and the actual vehicle position higher than 548.0303cm, and the threshold value $\phi($ 50 $)$ for the difference between the vehicle border and the obstacle border lower than 0.6864cm. The tool identifies the values 548.0303cm and 0.6864cm since, for the considered trace, they are respectively the maximum distance between the desired and the actual trajectory and the minimum distance between the vehicle and the obstacle border. This information shows to the engineer that (a) the vehicle is not precisely following the desired trajectory (the difference between the desired and the actual vehicle position should be increased) to satisfy the requirement, and (b) the vehicle is also not maintaining the distance from the obstacle (the difference between the vehicle border and the obstacle should be decreased to satisfy the requirement). However, since setting the value for the difference between the vehicle border and the obstacle to 0.6864cm satisfies the requirement, the diagnosis also shows that the vehicle does not collide with the obstacle. Changing the **and** logical operator into an **or** enables engineers to understand that making only one of the aforementioned changes makes the requirement satisfied.

SBTD can be customized depending on the type of diagnosis the engineers are looking for. The definition of the diagnosis influences the behavior of the Generator of Mutations and the Diagnostic Generator components. In the first place, the Generator of Mutations should generate requirements that most likely guide the search toward the generation of a

suitable diagnosis. Then, the Diagnostic Generator should aggregate the pairs produced by the Trace Checker based on the type of the desired diagnosis.

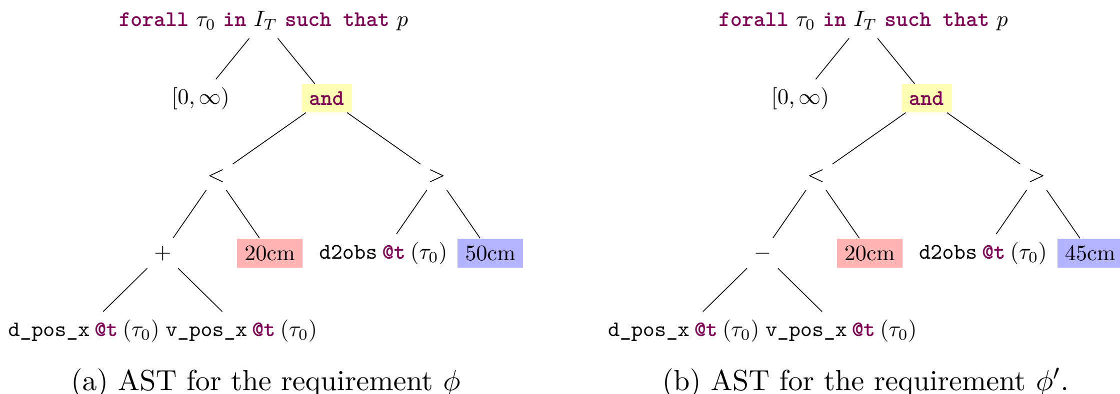
3.3 Search-based Trace Diagnostic for HLS

In this section, we describe an SBTD that supports requirements expressed in HLS. We present change-driven diagnosis (Section 3.3.1), the type of diagnosis supported by our SBTD instance. We describe the Generator of Mutations (Section 3.3.2), Trace-Checker (Section 3.3.3), and Diagnostic Generator (Section 3.3.4) components that support this type of diagnosis.

Table (3.1) Mutation Operators: the table contains the original formula and the mutated formula

OP	Original Formula	Mutated Formula
OP1	$\texttt{p}$	$\texttt{not p}$
OP2	$\texttt{tm}_1 \oplus \texttt{tm}_2$	$\texttt{tm}_1 \oplus' \texttt{tm}_2$
OP3	$\texttt{not p}$	$\texttt{p}$
OP4	$\texttt{p}_1 \ominus \texttt{p}_2$	$\texttt{p}_1 \ominus' \texttt{p}_2$
OP5	$\triangleright\, \tau$ $\texttt{in}$ I_T $\texttt{such that p}$	$\triangleright'\, \tau$ $\texttt{in}$ I_T $\texttt{such that p}$
OP6	$\triangleright\, \sigma$ $\texttt{in}$ I_J $\texttt{such that p}$	$\triangleright'\, \sigma$ $\texttt{in}$ I_J $\texttt{such that p}$
OP7	$\triangleright\, \rho$ $\texttt{such that p}$	$\triangleright'\, \rho$ $\texttt{such that p}$
OP8	$\texttt{tt}_1 \odot \texttt{tt}_2$	$\texttt{tt}_1 \odot' \texttt{tt}_2$
OP9	$\texttt{it}_1 \odot \texttt{it}_2$	$\texttt{it}_1 \odot' \texttt{it}_2$
OP10	$\texttt{vt}_1 \odot \texttt{vt}_2$	$\texttt{vt}_1 \odot' \texttt{vt}_2$
OP11	t	t'
OP12	j	j'
OP13	x	x'
OP14	s $\texttt{@i it}$	s' $\texttt{@i it}$
OP15	s $\texttt{@t tt}$	s' $\texttt{@t tt}$

$\oplus' \in \{>, <, \leq, \geq, =, \neq\} \setminus \{\oplus\}$
$\ominus' \in \{\texttt{or}, \texttt{and}, \texttt{implies}\} \setminus \{\ominus\}$
$\odot' \in \{+, -, *, /\} \setminus \{\odot\}$;
if $\triangleright = \texttt{forall}$, then $\triangleright' = \texttt{exists}$ if $\triangleright = \texttt{exists}$, then $\triangleright' = \texttt{forall}$.
$t, t' \in \mathbb{T}$, $j, j' \in \mathbb{J}$, $x, x' \in \mathbb{R}$, $\tau \in TV$, $\sigma \in SV$, $\rho \in RV$, $s, s' \in S$.



(a) AST for the requirement ϕ (b) AST for the requirement ϕ'.

Figure (3.3) AST of an HLS requirement and its mutation obtained by applying the mutation operator **OP13** to the node with the blue background.

[1]For simplicity, in our running example, we removed the "implies" operator from the mutation.

3.3.1 Change-Driven Diagnosis

Change-driven diagnosis explains requirements violations by describing which (set of) change(s) can lead to a requirement satisfied by the trace. For example, the decision tree (DT) reported in Figure 3.2 explains to engineers which changes applied to the requirement ϕ make it satisfied by the trace. This information helps engineers understand that, although the 50cm safety distance is violated and that the car does not follow the desired trajectory with a tolerance of 20cm, the car does not collide with the obstacle and the car deviates from the desired trajectory by a few meters: Setting the threshold value 0.68cm as safety distance between the car and the vehicle and 548.04cm as the tolerated deviation from the desired trajectory will make the requirement satisfied over the trace.

As our running example shows, engineers can select sub-portions of the requirements the changes should target. For example, for the requirement ϕ from Chapter 2 the red, yellow, and blue labeled boxes identify the sub-portions of the formula that the changes should refer to. The engineer is interested in how changes affect the satisfaction of the requirement, regarding: (i) the threshold distance between the desired and the actual trajectory of the car ($\phi($ 20 $)$), (ii) the threshold distance between the vehicle and the obstacle ($\phi($ 50 $)$), and (iii) the logical operator "**and**" ($\phi($ **and** $)$) that relates (i) and (ii). Intuitively, changes in the distances enable engineers to understand how the distance between the desired and the actual trajectory of the car and between the vehicle and the obstacle affect requirement satisfaction; changes in the logical operator "**and**" enable engineers to understand if both clauses of the requirements are violated.

> **Definição 1** [Change-Driven Diagnosis] *Let $\langle \pi, \phi \rangle$ be a trace-requirement combination made by a requirement (ϕ) unsatisfied over the trace (π) and sub(ϕ) a portion of the requirement the changes should target. A* diagnosis *d is a (set of) change(s) in the portion sub(ϕ) of the requirement ϕ that makes the requirement ϕ satisfied by π.*

An in-depth perspective of the SBTD steps to generate change-driven diagnosis for HLS requirements follows.

3.3.2 Generator of Mutations

This component receives a (set of) requirement(s) as inputs and generates a set of mutated requirements by sequentially performing the mutation and crossover operations.

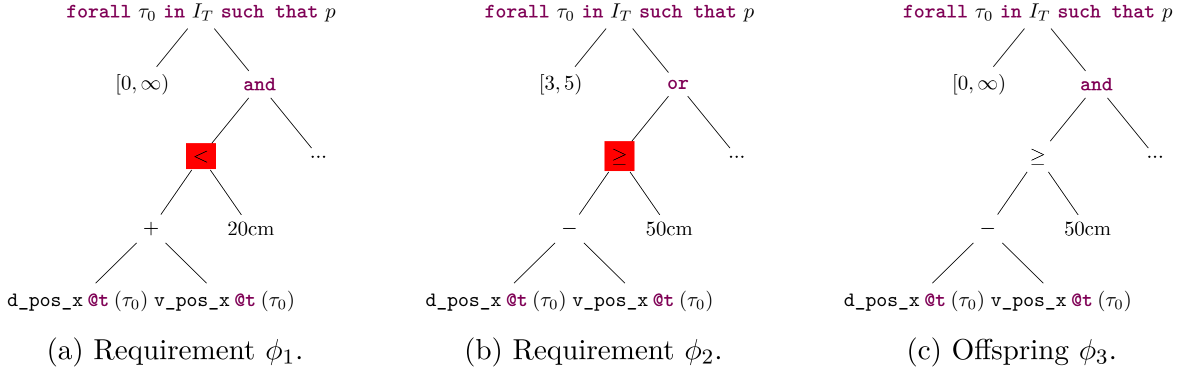
The *mutation operations* component considers an HLS requirement and changes the portions of the Abstract Syntax Tree (AST) that refer to the sub-portions of the requirements identified by the engineers. For example, the AST for the requirement ϕ of our motivating example is presented in Figure 3.3a. The portions of the abstract syntax tree

(AST) referring to the sub-portions of the requirements identified by the engineers are identified by colored nodes. Specifically, the nodes referring to the logical operator "**and**" and the threshold values 20cm and 50cm are with yellow, red, and blue background colors. The operator has to select the number of nodes to mutate between zero and the total number of nodes of the AST. This selection is related to portions of the requirement that the engineers are interested in. Then, it uses the mutation operators from Table 3.1 to mutate the nodes of the AST. Depending on the specific application, engineers can specify a subset of operators to be used by the generator of mutations. Operator **OP1** mutates the HLS requirement `p` into its negation **not** `p`. Operator **OP2** mutates the relational operator $\oplus$ by selecting another relational operator $\oplus'$. Operator **OP3** removes the negation operator from the HLS requirement **not** `p`. Operator **OP4** mutates the Boolean operator $\ominus$ used to combine the two requirements $\mathtt{p}_1$ and $\mathtt{p}_2$ by selecting another Boolean operator $\ominus'$. The operators **OP5**, **OP6**, and **OP7** mutate the existential quantifier **exists** into the universal quantifier **forall** and vice versa. The operators **OP8**, **OP9**, and **OP10** mutate the arithmetic operator $\odot$ by selecting another arithmetic operator $\odot'$. The operators **OP11**, **OP12**, and **OP13** mutate the time, index and value terms t, j, and x by selecting new values t', j', and x'. Finally, the operators **OP14** and **OP15** mutate the value terms s **@i** `it` and s **@t** `tt` into s' **@i** `it` and s' **@t** `tt` by selecting a new signal s'. All the mutation operators do not change the structure of the AST of the formula, but only the content of its nodes. In our running example, engineers select the operators **OP4**, that can mutate the logical operator "**and**", and the operator **OP13** that can mutate the value terms representing the threshold values 20cm and 50cm. Figure 3.3b presents the AST of the requirement ϕ': An example of a mutation for the AST from Figure 3.3a of the requirement ϕ of our running example where the operator **OP13** replaces the value "50cm" with the value "45cm".

ID	Parameter	Textual Description
CR	Crossover rate	Probability of applying the crossover operator.
MR	Mutation rate	Probability of applying the mutation operator.
PS	Population size	Number of requirements considered by the SBTD framework at each iteration.
SA	Selection Algorithm	The algorithm to be chosen for the selection of the requirements (Elitism or Roulette Wheel).
PTBC	Parents to Be Chosen	Number of requirements to be considered as a parent when using Elitism.
MG	Max Generation	Maximum number of iterations in the SBTD.
TS	Tournament Size	Number of requirements that compete to be selected as a parent.
TCTO	Trace check time out	Maximum time allowed for trace check to check a requirement.
PGTO	Program time out	Maximum time allowed for SBTD to find the requested solution.

Table (3.2) Configuration Parameters for our SBTD framework.

The *crossover operator* generates new candidate requirements by (a) selecting a pair of requirements, and (b) combining them. Next, we further explain how our algorithm selects the best pair of requirements by finding the best alignment between requirements,



(a) Requirement ϕ_1. (b) Requirement ϕ_2. (c) Offspring ϕ_3.

Figure (3.4) Example of application of the over operator: the requirement ϕ_3 is obtained from the requirement ϕ_1 by swapping the subtree with the root node with a red background with the corresponding subtree from the requirement ϕ_2.

then, we distill how we combine the best pair of requirements by swapping corresponding nodes from the AST trees representing each requirement.

To select the best requirements pair, the crossover operator computes a fitness value for each mutated requirement. The fitness value of each mutated requirement is obtained by comparing the mutated requirement with the original requirement using the score function of a pairwise alignment algorithm, namely Smith–Waterman [28]. Therefore, the fitness value is the score of the best local alignment between requirements such that the higher the fitness the more similar the mutated requirement to the original requirement is. Our choice for rewarding the similarity between requirements is grounded in the idea that fewer, but relevant, mutations in the requirements lead to fewer interactions between term changes, and consequently reduce the effect of the confounding bias [29]. In other words, the higher the similarity between the originally violated and the mutated HLS requirements, the lower the chances of having spurious factors that could incorrectly imply causation between the term changes and the requirement satisfaction (or violation).

Figure 3.5 demonstrates an example of calculating the fitness value of the mutated requirement ϕ' using the score function of the Smith-Waterman algorithm. The algorithm compares the distance between the original requirement (ϕ) and the mutated requirements (ϕ'), term by term. We use the algorithm as follows[2]: (i) mapping terms, (ii) calculating the initial scoring matrix, and (iii) collecting the score.

(i) *Mapping terms.* The algorithm maps the terms that can be mutated from both requirements (ϕ and ϕ') enriched with a "null" element to rows and columns of a scoring matrix. For example, Figure 3.5 represents the scoring matrix SM associated with the requirements ϕ and ϕ', where the terms that can be mutated in ϕ (i.e., $\phi($ 20 $)$, $\phi($ **and** $)$,

[2]Originally the SW algorithm computes the best local alignment to find the places where the term changes. However, for the purpose of informative diagnosis generation, we are concerned not only with the places of change but also with the domain and range of the values where the change takes place. Such step of our approach is further explained in Section 3.3.4.

	null	$\phi($ 20 $)$	$\phi($ and $)$	$\phi($ 50 $)$
null	0	0	0	0
$\phi'($ 20 $)$	0	3	1	0
$\phi'($ and $)$	0	1	6	4
$\phi'($ 45 $)$	0	0	4	3

Figure (3.5) Example of fitness calculation using the score function of the Smith-Waterman algorithm.

$\phi($ 50 $))$ and ϕ' (i.e., $\phi'($ 20 $)$, $\phi'($ and $)$, $\phi'($ 45 $))$, are respectively reported in the headers of its rows and columns.

(ii) *Calculating the scoring matrix.* The value zero is associated with matrix cells from rows and columns labeled with the "null" elements. The values of the remaining cells are calculated according to Equation (3.1).

$$SM[i,j] = max \begin{cases} SM[i-1,j-1] + s(\phi_i, \phi'_j) \\ SM[i-1,j] + W \\ SM[i,j-1] + W \\ 0 \end{cases} \tag{3.1}$$

The equation specifies that the value of the scoring matrix SM in position i, j, i.e., $SM[i,j]$, depends on the similarity score $(s(\phi_i, \phi'_j))$ of requirement ϕ in position i and requirement ϕ' in position j, with gap score (W, a.k.a. penality gap). The gap score penalizes formulae that require swapping many terms to be aligned. We set the similarity score $s(\phi_i, \phi'_j)$ to the value 3 when the terms from the column headers i and j coincide, to the value -3 otherwise. We considered the value -2 for the gap score (W).

(iii) *Collecting the score and measuring the fitness.* The score is the highest value in the scoring matrix. In the example from Figure 3.5, the score is 6 from cell $SM[3,3]$. Ultimately, we use the score as the fitness value in the following steps of the algorithm.

We implemented two selection methods that use these fitness values:

1. Elitism [30, 31]: selects two parents randomly between the best ten formulas following their *fitness*.

2. Roulette wheel [30, 31]: selects two parents based on their fitness, where the higher the *fitness*, the higher the probability of being selected.

To combine the pair of HLS requirements, we randomly select a node from the AST of the first requirement and swap it with the corresponding node of the second requirement. For example, Figure 3.4 presents an example of an application of the mutation operator: The mutation operator selects the sub-tree from the requirement ϕ_1 with the red node as

a root (Figure 3.4a) and swaps it with the corresponding sub-tree from the requirement ϕ_2 (Figure 3.4b) leading to requirement ϕ_3 (Figure 3.4c). Notice that, since the mutation operators do not change the structure of the AST, all the requirements have the same structure.

Table 3.2 lists the set of parameters to be configured by engineers to run the SBTD framework. For example, the crossover rate (CR) is the probability of applying the crossover operator.

The *Generator of Mutations* component generates a set of candidate requirements Ψ, which are then considered by the *Trace checker* component, explained as follows.

For the generator of mutations (1), we developed a Python script (i.e., ga.py) that implements the algorithm from Section 3.3.2. We decided to implement this procedure (instead of using an external library) since this decision enables controlling the data structures used by the algorithms to represent HLS requirements. This decision simplified the implementation of the operators from Table 3.1 and the fitness metric from Section 3.3.2.

3.3.3 Trace-Checker

The trace checker component considers the trace π and the candidate requirements Ψ and verifies which requirements hold on π. This is done by considering each HLS requirement $\phi \in \Psi$, and by running an existing trace-checker that can verify whether the requirement holds or not on the trace π, i.e., whether $\pi \models \phi$.

The *Trace Checker* component produces a set Δ of pairs $\{\langle \pi, \phi' \rangle, \upsilon\}$ made by the trace π, the mutated requirement ϕ', and the corresponding trace checking verdict υ. These pairs are fed into the *Diagnostic Generator*.

For the trace checker component (2), we used the `ThEodorE` [1] trace-checking tool since it supports requirements expressed in HLS. `ThEodorE` can produce three possible verdicts: "*satisfied*", if the trace satisfies the requirement, "*violated*", if it does not, or "*unknown*", if the SMT solver used by `ThEodorE` to solve the trace-checking problem can not deduce whether the requirement is satisfied or violated. The "*unknown*" verdict is returned when the underlying SMT technology used by the solver can not produce results for some specific instances of the problem [12]. Therefore, the diagnostic generator component will also create leaves labeled with the "*unknown*" verdict to explain cases where the trace-checker could not produce any verdict.

3.3.4 Diagnostic Generator

The *Diagnostic Generator* relies on two steps: (a) requirement filtering, and (b) decision-tree computation.

```
1 forall τ0 in IT such that p,[0,∞),and,<,+,...,satisfied
2 forall τ0 in IT such that p,[3,5),and,≥,-,...,violated
```

Figure (3.6) Entries considered by the learning algorithm.

The *requirement filtering* step selects the requirement mutations that are more similar to the original requirement for the computation of the decision tree while ensuring that the number of satisfied and unsatisfied requirements is the same. The requirement filtering ranks the mutated properties using the score function of the Smith–Waterman algorithm (as done by the cross-over operator — Section 3.3.2) for selecting the requirements to be combined. Then, it selects a subset of requirement mutations with the highest fitness values. The number of selected requirement mutations is defined by the parameter Parents to Be Chosen (PTBC) specified by the user (see Table 3.2).

The *decision-tree* computation works in two steps: *Data Preparation* and *Learning*

Data Preparation – Before running our learning technique, we have to prepare our data. Specifically, we have to represent the AST of each requirement in a format that is processable by a learning technique. We remark that the generator of mutations creates properties by not changing the structure of the AST.

Learning – The learning algorithm processes the input file and classifies the requirements based on the trace-checking verdict (satisfied or violated). We run J48 [32], a widely used ML algorithm [33] that generates decision trees that classify training data. Figure 3.2 illustrates an example of a resulting decision tree where $\phi($ **and** $)$ is the root node of the tree since splitting the $\phi($ **and** $)$ operator renders a bigger information gain than a split in $\phi(50)$, $\phi(20)$. Leaf nodes (*True*, *False*) are labeled with the frequency of whether the selected term results in the verdict.

For the diagnostic generator component (3), we used the Java implementation of the C4.5 algorithm [32] available in Weka [34]. We selected the C4.5 algorithm, since it is a widely used learning algorithm for decision trees [33], and Weka, since it is a well-known library of machine learning algorithms [35].

Chapter 4

Evaluation and Discussion

This chapter shows how we evaluate the approach presented. Here, we will define both the methodology and its results. Therefore, our evaluation assesses the trustworthiness of SBTD in identifying the correct cause for violated requirements. To this end, we consider two evaluation questions.

EQ1: How *accurate* is SBTD in producing diagnoses? (Section 4.2)

To answer this question, we validate the outcome from SBTD with a domain expert, which is regarded the ground truth of the experiments we conduct.

EQ2: How *efficient* is SBTD in producing informative diagnoses? (Section 4.3)

To answer this question, we assess the time required by SBTD to produce the diagnoses and assess the execution time of the components from Section 3.3.

To answer our questions we used `Diagnosis` as an instance of an SBTD framework. Our answers are based on the following: benchmark, experimental settings, and tool configuration of `Diagnosis`. Our `Diagnosis` tool has been implemented and is publicly available [21]. An appendix with a complete analysis of each experiment is also publicly available on Zenodo [36].

4.1 Experiment Setting and Tool Configuration

We considered 17 trace-requirement combinations, made by a trace and a requirement violated by the trace. Out of these combinations, 16 trace-requirement combinations were generated by considering 16 requirements from the ARCH 2023 Competition [11], an international SBST competition for Simulink models. The ARCH 2023 competition builds on previous editions to establish a consistent benchmark for falsifying temporal logic requirements over Cyber-Physical Systems (CPS). Its models and requirements have been

validated over several years, forming a robust foundation for evaluating system behavior across diverse scenarios. The requirements used are a comprehensive set of temporal logic specifications designed to test the resilience and robustness of a system, providing insights into its ability to maintain safety and functionality under varied conditions. This aligns well with our objective to evaluate how effectively our tool understands and responds to requirements in different scenarios.

The use of the ARCH benchmark is particularly relevant in our context because it offers a well-established baseline for testing CPS models. The requirements are designed not only to stress test the system but also to explore edge cases, making them ideal for assessing the nuanced impact of different requirement modifications. These benchmarks allow us to contextualize our tool's performance within a broader research landscape and identify areas of strength or improvement.

It is important to note, however, that our evaluation diverges from the primary goals of the ARCH competition. While the competition focuses on the falsification of requirements, our approach is centered on diagnosing requirement violations and understanding the underlying causes. This difference in focus means that direct comparisons between our tool and the competition results are not feasible. Lastly, the additional trace-requirement combination we used was derived from a recent example by Zhao et al. [37], which involves a robot following a trajectory while avoiding collisions, further enriching the evaluation context by incorporating real-world application scenarios.

The trace-requirement combinations from the ARCH 2023 Competition [11] were extracted from the replication package of two of the tools that participated in the competition (`ARIsTEO`[3] and `ATheNA-S` [4, 38]), and by considering a trace that violates the requirement that was returned by one of the tools. The traces have a large number of records (*min*=1594, *max*=10001, *Avg*=6565.3, *StdDev*=2656.9). Table 4.1 contains a textual description of the requirements we considered in our evaluation. The column ID reports the identifier from the ARCH 2023 competition. Out of the seven models used in the competition, we considered only the Automatic Transmission (AT) and Chasing Cars (CC) since they have the highest number of requirements. The requirement identifiers from the AT and CC models start with "AT" and "CC". For the robotic scenario, we considered one trace-requirement combination (RR). The requirement [39] specifies that the robot should follow a desired trajectory while avoiding collisions.

Since the requirements from the ARCH competition are formalized in Signal Temporal Logic (STL) [40], and `Diagnosis` supports HLS, we proposed an alternative specification of the requirements in HLS. Table 4.2 contains the HLS formalization for the requirements from Table 4.1. Since HLS is more expressive than STL [12], all the requirements could be expressed in HLS.

ID	Textual description
AT1	The vehicle's speed (v) shall be lower than 120 mph ($v \leq 120mph$) within [0 , 20]s.
AT2	The engine speed (ω) shall be lower than 4750 rpm ($\omega \leq 4750rpm$) within [0 , 10]s.
AT51	If the transmission enters Gear 1 within the time interval [0 , 30]s, it shall remain in that gear for the next 2.5 s.
AT52	If the transmission enters Gear 2 within the time interval [0 , 30]s, it shall remain in that gear for the next 2.5 s.
AT53	If the transmission enters Gear 3 within the time interval [0 , 30]s, it shall remain in that gear for the next 2.5 s.
AT54	If the transmission enters Gear 4 within the time interval [0 , 30]s, it shall remain in that gear for the next 2.5 s.
AT6a	If the engine speed is lower than 3000 rpm within [0, 30]s, then the vehicle speed shall be lower than 35 mph within [0, 4]s.
AT6b	If the engine speed is lower than 3000 rpm within [0, 30]s, then the vehicle speed shall be lower than 50 mph within [0, 8]s.
AT6c	If the engine speed is lower than 3000 rpm within [0, 30]s, then the vehicle speed shall be lower than 65 mph within [0, 20]s.
AT6abc	The requirements AT6a, AT6b, and AT6c shall be simultaneously satisfied. (Same mutation parameters as AT6c)
CC1	Car 5 shall always be at most 40 m ahead of car 4 within [0 , 100]s
CC2	Within [0 , 70]s, car 5 shall be at least 15 m ahead of car 4 at least once for the next [0 , 30]s.
CC3	At all times within $[0, 80]$s, for the next $20s$, car 2 shall always precede car 1 by at most $20m$, or car 5 shall precede car 4 by $40m$ at least once.
CC4	At all times within $[0, 65]$s, at least once in the next $30s$, car 5 shall always be at least 8 m ahead of car 4 for the next $5s$.
CC5	Within $[0, 72]$s, at least once in the next $8s$, if car 2 precedes car 1 by more than 9 m for $5s$, then car 5 shall precede car 4 by more than 9 m in the next $15s$.
CCx	Within [0 , 50]s, all cars shall always be at least 7.5 m ahead of the car immediately behind it. (The mutation operator is applied only for the distance between cars 4 and 5).
RR	From the beginning (time 0) to the end (∞) of the simulation, the following two conditions should hold: the difference (`d_pos_x` `@t` (τ_0) $-$ `v_pos_x` `@t` (τ_0)) between the desired position (`d_pos_x`) and the actual robot position (`v_pos_x`) in the x-axis at time τ_0 is lower than a threshold value (20 cm), and the Euclidean distance (`d2obs`) between the robot's border and the obstacle's border is greater than the threshold value (50 cm).

Table (4.1) Requirements from our benchmark.

For each trace-requirement combination, we defined the terms from the requirements that should be considered to understand the causes of the violations. The parts of the requirements and their formalization considered to understand the cause of the violations are colored in Table 4.1 and Table 4.2. We performed two experiments for each trace-requirement combination, considering different subsets of terms to be mutated. The two columns of Table 4.4 report the subset of terms considered for each trace-requirement combination. Considering two subsets of terms to be mutated for each trace-requirement combination led to 34 experiments (17×2) marked in Table 4.4 with the identifiers *exp1*, *exp2*, ..., *exp34*. The mutation operators to be used for each experiment and the value ranges to be considered to mutate the values of the real-valued variables are reported in Table 4.4. For example, for the requirement AT1 and experiment *exp1* the tool operator considered the mutation operator **OP13** for changing AT1(120) with threshold values of [100, 140]mph; for experiment *exp2* the operator considered the mutation operators

ID	HLS formalization
AT1	forall τ_0 in [0 , 20] such that v@t(τ_0) $\leq$ 120 .
AT2	forall τ_0 in [0 , 10] such that ω@t(τ_0) $\leq$ 4750 .
AT51	forall σ_0 in [t2i(0)+1,t2i(30)] such that ((gear@i(σ_0-1) $\neq$ 1) and (gear@i(σ_0) = 1)) implies (forall τ_0 in [i2t(σ_0), i2t(σ_0)+ 2.5] such that (gear @t(τ_0) = 1)).
AT52	forall σ_0 in [t2i(0)+1,t2i(30)] such that ((gear@i(σ_0-1) $\neq$ 2) and (gear@i(σ_0) = 2)) implies (forall τ_0 in [i2t(σ_0), i2t(σ_0)+ 2.5] such that (gear @t(τ_0) = 2)).
AT53	forall σ_0 in [t2i(0)+1,t2i(30)] such that ((gear@i(σ_0-1) $\neq$ 3) and (gear@i(σ_0) = 3)) implies (forall τ_0 in [i2t(σ_0), i2t(σ_0)+ 2.5] such that (gear @t(τ_0) = 3)).
AT54	forall σ_0 in [t2i(0)+1,t2i(30)] such that ((gear@i(σ_0-1) $\neq$ 4) and (gear@i(σ_0) = 4)) implies (forall τ_0 in [i2t(σ_0), i2t(σ_0)+ 2.5] such that (gear @t(τ_0) = 4)).
AT6a	(forall τ_0 in [0, 30] such that ω@t(τ_0) < 3000) implies (forall τ_1 in [0, 4] such that v@t(τ_1) < 35).
AT6b	(forall τ_0 in [0, 30] such that ω@t(τ_0) < 3000) implies (forall τ_1 in [0, 8] such that v@t(τ_1) < 50).
AT6c	(forall τ_0 in [0, 30] such that ω@t(τ_0) < 3000) implies (forall τ_1 in [0, 20] such that v@t(τ_1) < 65).
AT6abc	((forall τ_0 in [0,30] such that ω@t(τ_0) < 3000) implies (forall τ_1 in [0,4] such that v@t(τ_1) < 35)) and ((forall τ_2 in [0,30] such that ω@t(τ_2) < 3000) implies (forall τ_3 in [0,8] such that v@t(τ_3) < 50)) and ((forall τ_4 in [0, 30] such that ω@t(τ_4) < 3000) implies (forall τ_5 in [0, 20] such that v@t(τ_5) < 65)).
CC1	forall τ_0 in [0 , 100] such that ((y5@t(τ_0) - y4@t(τ_0)) $\leq$ 40).
CC2	forall τ_0 in [0 ,70] such that (exists τ_1 in [τ_0+ 0 ,τ_0+30] such that ((y5@t(τ_1) - y4@t(τ_1)) > 15).
CC3	forall τ_0 in [0,80] such that ((forall τ_1 in [τ_0,τ_0+20] such that ((y2@t(τ_1) - y1@t(τ_1)) < 20)) or (exists τ_2 in [τ_0,τ_0+20] such that ((y5@t(τ_2) - y4@t(τ_2)) > 40))).
CC4	forall τ_0 in [0,65] such that (exists τ_1 in [τ_0,τ_0+30] such that (forall τ_2 in [τ_1,τ_1+5] such that ((y5@t(τ_2) - y4@t(τ_2)) > 8))).
CC5	forall τ_0 in [0,72] such that (exists τ_1 in [τ_0,τ_0+8] such that ((forall τ_2 in [τ_1,τ_1+5] such that ((y2@t(τ_2) - y1@t(τ_2)) > 9)) implies (forall τ_3 in [τ_1+5,τ_1+20] such that ((y5@t(τ_3) - y4@t(τ_3)) > 9)))).
CCx	(forall τ_0 in [0 , 50] such that ((y5@t(τ_0) - y4@t(τ_0)) > 7.5)) and (forall τ_1 in [0,50] such that ((y4@t(τ_1) - y3@t(τ_1)) > 7.5)) and (forall τ_2 in [0,50] such that ((y3@t(τ_2) - y2@t(τ_2)) > 7.5)) and (forall τ_3 in [0,50] such that ((y2@t(τ_3) - y1@t(τ_3)) > 7.5)).
RR	forall τ_0 in [0,∞] such that ((d_pos_x @t(τ_0) - v_pos_x @t(τ_0)) < 20 and d2obs @t(τ_0) > 50).

Table (4.2) HLS formalization for the requirements from 4.1.

OP11, with value ranges of [0, 10]s for AT1(0) and [10, 30]s for AT1(20), and **OP13**, with value range of [100, 140]mph for AT1(120).

To answer the research questions of the evaluation, we configured `Diagnosis` as detailed in Section 4.1. We set 0.95 as a value for the crossover rate (CR) as done in a recent work [41]. Unlike Nunez et al. [41], who considered 0.10 as a value for the mutation rate (MR), we selected 0.90 to favor the generation of new mutations. The population size is set to 50 properties. We used the roulette wheel as a selection algorithm (SA), as done in a recent work [42]. We set 10 as a value for the parents to be chosen (PTBC) parameter. We set the value of the population size (50) for the tournament size (TS). The maximum number of generations (MG) is configured to stop the search when `Diagnosis` finds 1000 satisfied over the trace. We set a timeout of one hour for the trace-checking activity (TCTO). `Diagnosis` stops if it can not produce a diagnosis within five days (PGTO).

We executed experiments on a large computing platform with 1109 nodes, 64 cores, memory 249G or 2057500M, CPU 2 x AMD Rome 7532 2.40 GHz 256M cache L3.

Parameter	Value	Parameter	Value
CR	0.95	PTBC	10
MR	0.90	MG	1000 satisfied prop.
PS	50	TS	50
TCTO	1 hour	SA	Roulette Wheel
PGTO	5 days		

Table (4.3) Values for the configuration parameters of `Diagnosis` from 3.2.

4.2 Accuracy - EQ1

Our research hypothesis is that SBTD is effective in producing informative diagnoses. We assessed how accurate SBTD is in producing a diagnoses to validate our hypothesis. We compare diagnostics produced using `Diagnosis` to the causes that led to requirement violation, according to an expert.

Methodology. We compared diagnostics and predictions to answer whether SBTD is effective. `Diagnosis` generated diagnostics, an expert synthesized predictions for the requirements from Table 4.1. The comparison results from experiments with mutated operators, according to valid ranges.

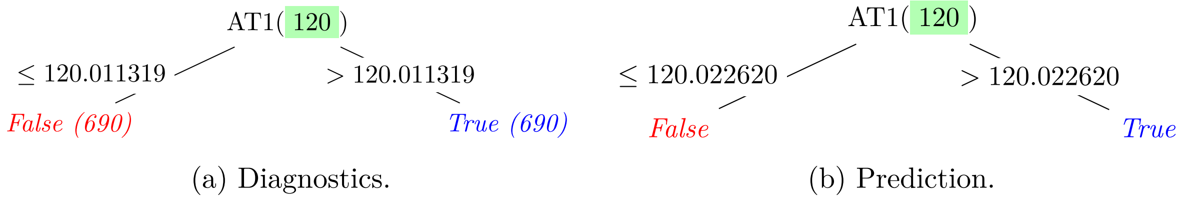
The experiment participants were the following: one played the role of the `Diagnosis` tool *operator* is the author of this thesis, and the other played the role of the *expert* is an authority on the AT and CC systems. Both participants did not exchange information about the experiments during the experimental set. The experimental set followed two steps: (i) cause derivation and (ii) diagnostics and prediction comparison. The experimental set is summarized in Table 4.4, which maps the requirement IDs to independent variables (namely Operators), and valid ranges exercised in each experiment. The colored background in Table 4.4 maps terms from the Table 4.1 to mutated operators.

(i) *Cause Derivation.* The *tool operator* and the *expert* worked separately to derive the causes of the violated requirements. The *tool operator* configured `Diagnosis` using the configuration parameters in Section 4.1. As a result, the *tool operator* collected one decision tree for each experiment. For example, Figure 4.1a reports the diagnosis for the experiment *exp1* that considers the impact of the value AT1(120) on the satisfaction of the requirement AT1. The decision tree shows that setting the value of AT1(120) higher and lower than 120.006093 respectively makes the property satisfied or violated since the signal reaches the value 120.006093. Note that the DT leaves contain the same number (1013) of satisfied and unsatisfied requirements since the requirement filtering step ensures that the number of satisfied and unsatisfied requirements is the same.

The *expert* analyzed the violated requirement according to their experience and manually synthesized a prediction. To synthesize the prediction, the *expert* plotted the trace

Req. ID	Exp.	Operators	Valid Range	Exp.	Operators	Valid Range
AT1	*exp1*	OP13	[100,140]mph	*exp2*	OP11 , OP11 , OP13	[0,10]s , [10,30]s , [100,140]mph
AT2	*exp3*	OP13	[4700,4800]rpm	*exp4*	OP11 , OP11 , OP13	[0,5]s , [5,15]s , [4700,4800]rpm
AT51	*exp5*	OP11	[0,5]s	*exp6*	OP11 , OP11 , OP11	[0,15]s , [15,45]s , [0,5]s
AT52	*exp7*	OP11	[0,5]s	*exp8*	OP11 , OP11 , OP11	[0,15]s , [15,45]s , [0,5]s
AT53	*exp9*	OP11	[0,5]s	*exp10*	OP11 , OP11 , OP11	[0,15]s , [15,45]s , [0,5]s
AT54	*exp11*	OP11	[0,5]s	*exp12*	OP11 , OP11 , OP11	[0,15]s , [15,45]s , [0,5]s
AT6a	*exp13**†	OP13 , OP13	[2800,3200]rpm , [30,40]mph	*exp14**†	OP11 , OP13 , OP11 , OP13	[20,40]s , [2800,3200]rpm , [2,6]s , [30,40]mph
AT6b	*exp15**†	OP13 , OP13	[2800,3200]rpm , [40,60]mph	*exp16**†	OP11 , OP13 , OP11 , OP13	[20,40]s , [2800,3200]rpm , [4,12]s , [40,60]mph
AT6c	*exp17**†	OP13 , OP13	[2800,3200]rpm , [50,80]mph	*exp18**†	OP11 , OP13 , OP11 , OP13	[20,40]s , [2800,3200]rpm , [15,25]s , [50,80]mph
AT6abc	*exp19**†	OP13 , OP13	[2800,3200]rpm , [50,80]mph	*exp20**†	OP11 , OP13 , OP11 , OP13	[20,40]s , [2800,3200]rpm , [15,25]s , [50,80]mph
CC1	*exp21*†	OP13	[30,50]m	*exp22*†	OP11 , OP11 , OP13	[0,50]s , [50,100]s , [30,50]m
CC2	*exp23**†	OP11	[0,20]s	*exp24**†	OP11 , OP11 , OP13	[0,20]s , [0,10]s , [12,18]m
CC3	*exp25**	OP5	`{forall,exists}`	*exp26**	OP5 , OP5 , OP4	`{forall,exists}` , `{forall,exists}` , `{and, or}`
CC4	*exp27**†	OP13	[6,10]m	*exp28**†	OP5 , OP5 , OP13	`{forall,exists}` , `{forall,exists}` , [6,10]m
CC5	*exp29**†	OP13 , OP13	[7,11]m , [7,11]m	*exp30**†	OP2 , OP13 , OP2 , OP13	{>, <} , [7,11]m , {>, <} , [7,11]m
CCx	*exp31*†	OP13	[5,10]m	*exp32*†	OP11 , OP11 , OP13	[0,25]s , [25,75]s , [5,10]m
RR	*exp33*	OP13 , OP4	[500,700]cm , `{and, or, implies}`	*exp34*	OP13 , OP4 , OP13	[500,700]cm , `{and, or}` , [0,2.5]cm

Table (4.4) Mutation operators (Operators) and ranges for the value terms (Ranges) of each experiment (Exp).



(a) Diagnostics. (b) Prediction.

Figure (4.1) Diagnostic and prediction for the experiment *exp1*.

and tried to reverse-engineer the cause of the violation and express it as a DT. For example, 4.1b reports the prediction for the experiment *exp1*. Note that, for this example, since the expert can inspect the trace, they can identify the exact condition (>120.022620) that turns the property from violated to satisfied.

(ii) Diagnostics and Prediction Comparison. We compared the diagnostics (*tool operator*'s decision trees) and the predictions (*expert*'s decision trees).

For our experiments, the DT produced by the tool operator and the expert can be significantly different: values can be considered in multiple orders and be split various times by each decision tree. Therefore, to compare these DTs we use an empirical approach inspired by the approach presented by Gaaloul et al. [27] originally used to compare software assumptions. The approach requires generating a set of properties by considering 101 assignments for each numerical value mutated by the SBTD algorithm. For example, for *exp2* a set of properties is generated by considering 101 assignments for each variables: for AT1(0) values from 0 to 10 with increments of 0.1, for AT1(20) values from 10 to 30 with increments of 0.2, and for AT1(120) values from 100 to 140 with increments of 0.4. Considering the combinations of these values leads to a total of 1 030 301 properties. When the mutations also involved logical operators (e.g., *exp26*) this procedure was replicated for all the possible assignments of the logical operators. For example, for *exp26* the procedure assigned CC3(**forall**), CC3(**forall**), CC3(**and**) to both {**forall**, **forall**, **and**}, {**forall**, **forall**, **or**}, and {**forall**, **exists**, **and**}, and all the remaining combinations of logical operators. For each property, we assessed whether the property was expected to be satisfied or violated according to the DTs produced by the tool operator and the expert. This was done by assessing whether the leaf of the DT associated with that formula was labeled with a *True* or a *False* value. A true positive (TP) is when the property is satisfied according to both the DTs (the one from the tool operator and the one from the expert). A true negative (TN) is when the property is violated according to both DTs. A false positive (FP) is when the property is satisfied by the DT returned by the tool operator and violated by the one produced by the expert. Finally, a false negative (FN) is when the property is violated by the DT returned by the tool operator and satisfied by the one produced by the expert. We analyzed the precision and recall of the method.

Note that the `ThEodorE` trace-checker returns that a property is violated by a trace

when Z3 confirms that the logical formula generated by the trace-checker is satisfiable; It returns that the property is satisfied in the opposite case. In our case, Z3 formula contains quantifiers, we empirically observed that Z3 usually takes longer to confirm the satisfiability of the logical formula, i.e., to show that a property is violated by a trace. Therefore, for some of our experiments in which the trace-checker could return that the property was satisfied by some traces but could not provide the opposite result (marked with an asterisk "*" in Table 4.4), we assume the property to be violated when the Z3 solver returned "*unknown*" result, assuming that for these instances the Z3 solver would have returned a "*satisfied*" verdict with more time available. Our results confirm the validity of this hypothesis for our experiments. Finally, for some of our experiments `Diagnosis` could not generate 1000 satisfied properties (see Table 3.2) within five days. For those cases (marked with an asterisk "†" in Table 4.4 and Section 4.3), we run the DT computation manually after `Diagnosis` ends.[1]

Results. Running our experiments would have required approximately 109 days. The time was reduced to five days by exploiting the parallelization facilities of our computing platform.

For 33 out of 34 experiments, the SBTD tool could produce a diagnosis within five days. The boxplot from Figure 4.2 presents the precision ($\frac{TP}{TP+FP}$) and recall ($\frac{TP}{TP+FN}$) of SBTD across the different experiments. SBTD shows a considerable precision (*min*=90.2%, *max*=100.0%, *Avg*=98.9%, *StdDev*=2.1%) across the different experiments showing that the value ranges for which the requirements are satisfied are confirmed by the expert. SBTD shows a considerable recall (*min*=54.6%, *max*=100.0%, *Avg*=92.7%, *StdDev*=12.3%) across the different experiments showing that SBTD can identify most of the values for which the requirements are satisfied.

For one out of 34 experiments (*exp25* — identified with a brown background in Table 4.4), the SBTD could not produce a diagnosis within five days. For this case, the `ThEodorE` trace-checker leads to the timeout of the SBTD tool. As reported by the authors [1, 12], while supporting an expressive logic (HLS), `ThEodorE` inherits the limitations of the SMT technology used to solve the trace-checking problem, which can require considerable time to solve the satisfiability problem and terminate with an "*unknown*" result. Note that we assumed that an "*unknown*" result confirmed the violation of a property only when for some of the generated trace-requirements combinations the trace-checker could confirm that the property was satisfied by the trace. This was not the case for *exp25*, where `ThEodorE` could never produce a trace-checking verdict.

[1]For *exp26* could not create a thousand mutations since there are only eight possible mutations of the original requirement.

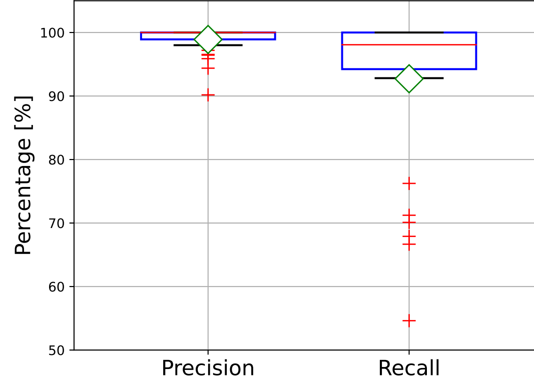


Figure (4.2) Precision and recall of SBTD across the different experiments. Diamonds depict the average, red lines are the median, and pluses depict the outliers.

EQ1 - Accuracy

The results show that our SBTD framework returned an accurate diagnosis for 33 out of 34 experiments. For one of our experiments, the performance limitations of the trace-checker we selected (`ThEodorE`) did not enable our SBTD framework to produce a diagnosis.

4.3 Efficiency - EQ2

We assessed how long SBTD takes in producing informative diagnoses as follows.

Methodology. We consider the experiments executed to answer section 4.2. We recorded the time `Diagnosis`, and its components (see Chapter 3), required to produce the diagnoses and analyzed it.

Results. Section 4.3 reports the total time required by each experiment as well as the time required by the generator of mutations [1], the trace-checker [2], and the diagnostic generator [3]. SBTD could produce a diagnosis within 47 hours (*min*=6.1h, *max*=46.7h, *Avg*=14.8h, *StdDev*=9.6h) for 14 experiments. This computational time is acceptable for many applications since it is negligible compared to the development time of the CPS. For 20 out of 34 experiments, SBTD could not generate 1000 satisfied requirements (see Table 3.2) within five days (120h). However, as discussed in Section 4.2, forcing the computation of the DT manually leads to accurate results even with fewer satisfied properties. Finally, for experiment *exp25* (labeled with the '-' character in Section 4.3) `Diagnosis` could not produce a diagnosis within 120h and we could not force its computation manually since `ThEodorE` did not produce a trace-checking verdict for any of the requirement mutations.

ID	Exp.	Tool (total)	Tool 1	Tool 2	Tool 3	Exp.	Tool (total)	Tool 1	Tool 2	Tool 3
AT1	*exp1*	11.6h	10.5min	11.4h	5.09s	*exp2*	8.8h	11.5min	8.6h	6.12s
AT2	*exp3*	7.9h	10.6min	7.7h	6.52s	*exp4*	6.1h	11.7min	5.9h	6.42s
AT51	*exp5*	12.5h	18.8min	12.2h	6.17s	*exp6*	16.1h	20.7min	15.7h	6.67s
AT52	*exp7*	12.5h	25.1min	12.3h	5.71s	*exp8*	46.7h	1.63h	45.0h	6.13s
AT53	*exp9*	22.1h	14.8min	21.9h	6.89s	*exp10*	13.0h	21.0min	12.7h	7.40s
AT54	*exp11*	16.5h	17.2min	16.2h	6.02s	*exp12*	11.5h	14.3min	11.2h	7.74s
AT6a	*exp13*†	120.0h	10.26s	111.3h	1.31s	*exp14*†	120.0h	4.15s	108.6h	2.34s
AT6b	*exp15*†	120.0h	3.56s	111.3h	2.02s	*exp16*†	120.0h	4.89s	105.0h	2.52s
AT6c	*exp17*†	120.0h	5.15s	104.0h	2.17s	*exp18*†	120.0h	3.87s	112.3h	2.50s
AT6abc	*exp19*†	120.0h	10.08s	118.6h	2.37s	*exp20*†	120.0h	19.29s	115.1h	2.95s
CC1	*exp21*†	120.0h	14.10s	113.8h	1.55s	*exp22*†	120.0h	32.85s	114.8h	2.84s
CC2	*exp23*†	120.0h	19.28s	117.3h	2.42s	*exp24*†	120.0h	5.86s	112.5h	2.19s
CC3	*exp25*†	120.0h	0.44s	81.5h	-	*exp26*	120.0h	0.44s	81.5h	1.24s
CC4	*exp27*†	120.0h	4.27s	91.9h	1.97s	*exp28*†	120.0h	3.75s	107.0h	2.06s
CC5	*exp29*†	120.0h	0.52s	92.9h	1.93s	*exp30*†	120.0h	2.21s	101.3h	2.32s
CCx	*exp31*†	-	-	-	-	*exp32*†	-	-	-	-
RR	*exp33*	6.4h	6.7min	6.2h	0.12s	*exp34*	4.6h	1.53min	4.6h	7.70s

Table (4.5) Time required by our SBTD tool to extract the diagnosis.

EQ2 - Efficiency

Our SBTD framework could produce a diagnosis within 47 hours for 14 of our experiments. For 20 experiments our SBTD did not terminate within five days since it could not generate 1000 satisfied requirements. For one of our experiments, `Diagnosis` could not produce a diagnosis.

4.4 Discussion and Threats to Validity

Our SBTD approach uses DT to express the diagnosis and inherits the limitations of this technology: The DT expresses conjunctions of conditions expressed by its node, and each node of the DT expresses a condition that only refers to one term of the formula. Therefore, we can not learn more complex relations between input signals like the quadratic relation between the upper temporal limit AT1(20) and the upper speed limit AT1(120) in *exp*2. We plan to address this limitation by considering other ML techniques (e.g., Genetic Programming) in the future.

SBTD is defined to support and complement the activity of expert designers. First, it can automatically synthesize a diagnosis without any human intervention. Second, it can also help experts by confirming their diagnostic witnesses since experts can be wrong or miss corner cases. Third, another advantage of `Diagnosis` is that its activity can be parallelized. While the expert activity is sequential and can not be parallelized, many instances of `Diagnosis` can be executed in parallel by analyzing different trace-requirements combinations. Finally, our results show that SBTD can produce accurate

results in a few days. Engineers can wait a few days for an informative diagnosis in many practical scenarios (e.g., safety-critical applications),

Internal Validity. We compared the diagnosis produced by `Diagnosis` with the one proposed by an expert. We remark that our expert has extensive knowledge about our benchmark models. Therefore, it is likely that they are producing accurate diagnoses.

For EQ1, the metric used to compare the DTs produced by the tool and the expert could threaten the internal validity of our results. For experiments in which only one value was mutated, we could have computed the error between the values identified by the expert and the tool as a metric for success. However, this metric would not apply to experiments with multiple values. Our approach enables us to consider these two cases seamlessly.

For EQ1 and EQ2, the values selected for the configuration parameters of our tool (Table 3.2) threaten the internal validity of our results. For example, the maximum number of generations (MG) and the usage of the J48 algorithm could have threatened the precision and recall of the SBTD procedure. To have a ballpark estimation of considering a lower value for MG on our results, we repeated *exp2* and *exp4* by setting the value of MG to 100 (instead of 1000). The precision and recall (EQ1) of the SBTD procedure for MG equal 1000 and 100 are comparable: for *exp2* changed respectively from 98.6% and 97.6% (MG=1000) to 96.4% and 89.8% (MG=100), for *exp4* changed respectively from 98.9% and 100.0% (MG=1000) to 100.0% and 100.0% (MG=100). The computation time (EQ2) of the SBTD procedure for *exp2* changed from 8.78h (MG=1000) to 1.12h (MG=100). The computation time (EQ2) of the SBTD procedure for *exp4* changed from 6.16h (MG=1000) to 0.86h (MG=100). While our experiments provide the results for a specific configuration (defined by selecting configuration values from the literature), in practice, engineers should configure the SBTD tool depending on their domain-specific needs and the desired precision and recall.

We selected `ThEodorE` as a trace-checking tool to implement our methodology since it supports complex signal logic specifications. Our experimentation confirms some of the limitations regarding the efficiency of this tool [12]. Specifically, in some of our experiments (e.g., *exp17*, *exp23*, *exp29*), the trace-checker could not provide a verdict within the allotted time. For these cases, the problem was the size of the instance the SMT solver had to consider. In the future, we plan to extend our framework to consider other trace-checking tools. Other trace-checkers (e.g., dp-Taliro [43]) are more efficient, but support less expressive languages.

The DTs defined by the experts threaten the internal validity of our results. First, we used the DTs provided by the expert as a ground truth. However, we are not sure that the prediction provided by the expert is correct. The only way to have a correct prediction

would have been to verify all the possible requirements with a trace-checker. However, this is impossible since (a) the properties are defined on real numbers (and therefore are infinite), and (b) considering a large subset of properties would have been computationally demanding (e.g., for *exp9* running the trace-checking tool for all the 1 030 301 properties would have required more than a year). Second, other engineers could have defined other DTs for our case studies. However, for experiments concerning requirements expressing invariants where a single value is to be considered in the diagnostic activity the procedure followed by the expert is not subjective: The expert defined the DTs by extracting the minimum and maximum values assumed by the signals. For the other requirements, the opinion of the expert penalizes our research. Our expert knows the models from which the traces are obtained and has inspected the traces. However, our expert is not the developer of these models and their opinion about the diagnosis may not be correct. Therefore, when there are mispredictions from the expert (i.e., false positives and false negatives) the expert could be wrong and the tool may produce the correct answer. Considering the opinions of other experts may reduce the number of false positives and negatives in our study.

The selection of HLS could threaten the internal validity of our results. Considering other languages (e.g., SB-TemPsy-DSL [44], Restricted Signals First Order Logic [45]) for specifying the requirements could lead to different results.

Although the *Generator of Mutations* is a stochastic algorithm that could provide different running times every run, we could not run our experiment multiple times due to limited computational resources. Running our experiment would have required approximately 109 days (reduced to five days by exploiting the parallelization facilities of our computing platform). However, running our experiments for different models and requirements mitigates this threat.

External Validity. The set of trace-requirement combinations we considered in our experiments could threaten the external validity of our results as considering other trace-requirement combinations may lead to different results. However, the requirements of our benchmark refer to different case studies and use different logical operators.

Overfitting and hyperparameter tuning could threaten the external validity of our results, i.e., the same configuration applied to other benchmarks could produce different results. However, our configuration is not experiment-specific: It is shared across all of our experiments including different models and requirements. Moreover, through the fitness function, we select the parameters that are more likely to explain the cause of the requirement violation across different models of our experiments. Through the informative diagnosis cycle, we identify the ranges of those parameters that are crucial for one to reason why the various requirements were violated. However, due to the stochastic nature of our

process, the diagnosis cycle is not guaranteed to find an optimal solution.

Chapter 5

Related Work

Property violations are typically explained by exploiting some notion of causality (e.g., [15, 46, 18]) to extrapolate the causes of the failure (e.g., an event A is said to be a cause of event B if, had A not happened then B would not have happened). These causes typically refer to portions of (a) the trace (e.g., portions of the trace), or (b) the property (e.g., portions of the property) responsible for the violation.

Approaches that extrapolate information coming from the *trace* typically isolate slices of the traces that contain the causes for the property violation (e.g., [47, 13, 14, 15, 16, 18]). Other approaches explain the property violation by checking for traces showing common behaviors that lead to the satisfaction and violation of the property (e.g., [20, 17]). Unlike these approaches, `Diagnosis` explains the violation by describing how mutations applied to the property lead to its satisfaction or violation.

Approaches that extrapolate information coming from the *property* (e.g., [19, 48]) typically exploit its structure to provide viable diagnoses. For example, pattern-based diagnostic approaches (e.g., [19]), enrich trace-checking verdicts (i.e., [44]) by exploiting the syntactical structure of the property (i.e., the patterns used to define the property of interest) to compute viable diagnoses. These approaches require engineers to define a predefined set of possible violation causes and corresponding diagnoses upfront or assume a library of violation causes and corresponding diagnoses to be available. Unlike these approaches, `Diagnosis` relies on a novel evolutionary approach that can dynamically generate new diagnoses by applying the mutation and cross-over operators.

Approaches that can explain property violations are also common within the context of model-checking. Most of the existing approaches (e.g., [49, 50, 51, 52, 53, 54, 55]) are based on deductive reasoning techniques that start from some initial assertions examine how logical operators support the conclusion that the property is violated. Other approaches extract information from the model (e.g., model slices) to explain the model-checking verdict (e.g., [56, 57, 58, 59, 48, 60, 61, 62, 63, 64]). Explainability was also studied in the

context of anomaly detection (see [65] for a recent survey). Recent work also considered how to explain spurious failures detected by test case generation frameworks [66] and via feature engineering [67]. Unlike these approaches, `Diagnosis` produces informative diagnosis in the context of the trace-checking problem domain, a significantly different problem. Additionally, `Diagnosis` relies on an evolutionary approach.

The literature on Mutation Testing (e.g., [68, 69, 70, 71, 72]) highlights the challenge of equivalent mutants — mutants that show no detectable behavioral changes compared to the original program. These mutants are indistinguishable from the original and undermine the mutation testing process by clouding the observable space. Our approach uses Genetic Programming to modify violated requirements, which could also encounter equivalent mutants. However, we mitigate these issues through: (i) User-Selected Terms for Mutation, (ii) Constrained Requirement Generation, and (iii) Information from No Impact. By allowing users to specify changeable terms, enforcing structural constraints, and treating unchanged output as valuable data, we ensure focused modifications and extract meaningful information even from equivalent mutants.

Chapter 6

Conclusion

In this work, we proposed a Search-Based Trace-Diagnostic (SBTD) technique, a novel trace-diagnostic approach designed to help engineers understand the cause of violations of Cyber-Physical System (CPS) requirements. This technique is based on evolutionary search, utilizing mutation, recombination, and selection to iteratively refine a set of candidate diagnoses. The evolutionary search is guided by a fitness function, which evaluates the quality of the identified solutions, allowing the technique to efficiently explore the space of possible diagnoses.

The SBTD technique does not require any formal system model to generate diagnoses, which reduces overhead and makes it practical for complex systems where constructing an explicit model would be challenging. Instead, it relies solely on a violated property and a system's trace, which makes it particularly well-suited for situations where constructing an explicit system model would be difficult or impractical. By applying evolutionary search to generate new candidate diagnoses, the SBTD approach offers a powerful mechanism to identify the most relevant changes that could explain requirement violations, even in complex CPS environments.

Our SBTD framework supports properties expressed using the Hybrid Logic of Signals (HLS), a highly expressive formalism that enables requirements to be described in terms of temporal relationships and signal properties. We developed a set of mutation operators specifically designed for HLS requirements, enabling the evolutionary algorithm to generate meaningful variations in the properties while maintaining logical consistency. This approach helps ensure that each mutation leads to a candidate requirement that remains similar enough to the original, thus providing more useful insights into the root causes of the observed violations.

The implementation of our SBTD framework is available as an open-source tool named `Diagnosis`, which takes signal-based temporal logic requirements (expressed in HLS) as input and produces a diagnosis that explains the observed violation. `Diagnosis` was rig-

orously evaluated in terms of both *accuracy* and *efficiency*, with an extensive empirical evaluation involving 17 trace-requirement combinations that led to property violations. The results of this evaluation demonstrated that `Diagnosis` is capable of generating trustworthy diagnoses, accurate within a practical time frame, thereby supporting engineers in the task of trace analysis for CPS.

Due to the novelty of the approach, we were unable to directly compare our technique with other existing methods from the literature. However, we performed a comparative analysis with expert predictions, and the results indicate that `Diagnosis` can produce diagnoses that are consistent with expert assessments in most experiments. This validates the trustworthiness of our approach in identifying the root causes of CPS requirement violations.

Our tool, `Diagnosis`, is publicly available, making it accessible for researchers and practitioners alike. It can also be extended to support properties specified in other logics, provided that a corresponding trace-checker is available. The generality of our approach, combined with the expressiveness of HLS, results in a powerful tool capable of diagnosing virtually any system. The SBTD framework can provide informative insights into the problems that occurred during the trace without requiring human intervention.

Nevertheless, `Diagnosis` is not without limitations. The lack of an explicit system model means that the tool cannot directly infer the specific point of failure within the system; instead, it relies on the operator to investigate that aspect. Additionally, the tool's performance is inherently tied to the efficiency of the trace-checker used. For some experiments, the given time budget was insufficient to produce results, highlighting the need for efficient trace-checkers to support rapid diagnostics.

In future work, we plan to experiment on different case studies, e.g., in the space domain, to check the generality of `Diagnosis`, as well as to better assess its accuracy and efficiency. We also plan to benchmark other similarity fitness functions and research a fitness function specifically targeted to CPS requirements. Additionally, we intend to experiment with languages different from HLS, such as SB-TemPsy-DSL or Restricted Signals First-Order Logic, to determine the extent to which the results in this thesis are confirmed. Finally, we also aim to research and study alternatives for the implementation of Diagnostic Generator (DG), such as Genetic Programming (GP), Machine Learning (ML), and other methods, and benchmark them to explore potential improvements.

References

[1] Menghi, Claudio, Enrico Viganò, Domenico Bianculli, and Lionel C Briand: *Theodore: A trace checker for cps properties.* In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 183–184. IEEE, 2021. x, 2, 9, 10, 20, 29

[2] Krueger, Charles W.: *Software reuse.* ACM Comput. Surv., 24(2):131–183, June 1992, ISSN 0360-0300. `https://doi.org/10.1145/130844.130856`. 1

[3] Menghi, Claudio, Shiva Nejati, Lionel Briand, and Yago Isasi Parache: *Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification.* In *International Conference on Software Engineering (ICSE)*, page 372–384. IEEE / ACM, 2020. 2, 23

[4] Formica, Federico, Fan Tony, and Claudio Menghi: *Search-based software testing driven by automatically generated and manually defined fitness functions.* ACM Transactions on Software Engineering and Methodology, 33(2), 2023. 2, 23

[5] Waga, Masaki: *Falsification of cyber-physical systems with robustness-guided black-box checking.* In *International Conference on Hybrid Systems: Computation and Control (HSCC)*. ACM, 2020. 2

[6] Zhang, Zhenya, Deyun Lyu, Paolo Arcaini, Lei Ma, Ichiro Hasuo, and Jianjun Zhao: *Effective Hybrid System Falsification Using Monte Carlo Tree Search Guided by QB-Robustness.* In *Computer Aided Verification*, pages 595–618. Springer, 2021. 2

[7] *NNFal.* `https://gitlab.com/Atanukundu/NNFal`, April 2023 [Online]. 2

[8] Peltomäki, Jarkko and Ivan Porres: *Requirement falsification for cyber-physical systems using generative models.* arXiv preprint arXiv:2310.20493, 2023. 2

[9] Annpureddy, Yashwanth, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan: *S-TaLiRo: A tool for temporal logic falsification for hybrid systems.* In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257. Springer, 2011. 2

[10] Thibeault, Quinn, Jacob Anderson, Aniruddh Chandratre, Giulia Pedrielli, and Georgios Fainekos: *PSY-TaLiRo: A Python Toolbox for Search-Based Test Generation for Cyber-Physical Systems.* In *Formal Methods for Industrial Critical Systems*, pages 223–231. Springer, 2021, ISBN 978-3-030-85248-1. 2

[11] Menghi, Claudio, Paolo Arcaini, Walstan Baptista, Gidon Ernst, Georgios Fainekos, Federico Formica, Sauvik Gon, Tanmay Khandait, Atanu Kundu, Giulia Pedrielli, *et al.*: *Arch-comp 2023 category report: Falsification.* In *International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH23)*, volume 96, pages 151–169, 2023. 2, 22, 23

[12] Menghi, Claudio, Enrico Viganò, Domenico Bianculli, and Lionel C Briand: *Trace-checking CPS properties: Bridging the cyber-physical gap.* In *International Conference on Software Engineering (ICSE)*, pages 847–859. IEEE/ACM, 2021. 2, 4, 8, 20, 23, 29, 32

[13] Ferrère, Thomas, Oded Maler, and Dejan Ničković: *Trace diagnostics using temporal implicants.* In *International Symposium on Automated Technology for Verification and Analysis*, pages 241–258. Springer, 2015. 2, 35

[14] Mukherjee, Subhankar and Pallab Dasgupta: *Computing minimal debugging windows in failure traces of ams assertions.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 31(11):1776–1781, 2012. 2, 35

[15] Beer, Ilan, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard Trefler: *Explaining counterexamples using causality.* Formal Methods in System Design, 40:20–40, 2012. 2, 35

[16] Ničković, Dejan, Olivier Lebeltel, Oded Maler, Thomas Ferrère, and Dogan Ulus: *Amt 2.0: qualitative and quantitative trace analysis with extended signal temporal logic.* International Journal on Software Tools for Technology Transfer, 22:741–758, 2020. 2, 35

[17] Dawes, Joshua Heneage and Giles Reger: *Explaining violations of properties in control-flow temporal logic.* In *International Conference on Runtime Verification (RV)*, pages 202–220. Springer, 2019. 2, 35

[18] Dou, Wei, Domenico Bianculli, and Lionel Briand: *Model-driven trace diagnostics for pattern-based temporal specifications.* In *International Conference on Model Driven Engineering Languages and Systems*, MODELS, page 278–288. ACM/IEEE, 2018, ISBN 9781450349499. 2, 35

[19] Boufaied, Chaima, Claudio Menghi, Domenico Bianculli, and Lionel C Briand: *Trace diagnostics for signal-based temporal properties.* IEEE Transactions on Software Engineering, 49(5):3131–3154, 2023. 2, 8, 35

[20] Luo, Qingzhou, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu: *Rv-monitor: Efficient parametric runtime verification with simultaneous properties.* In *International Conference on Runtime Verification (RV)*, pages 285–300. Springer, 2014. 2, 35

[21] *Diagnosis.* `https://github.com/Gastd/ga-hls/tree/main`, October 2024 [Online]. 5, 22

[22] Maler, Oded and Dejan Nickovic: *Monitoring temporal properties of continuous signals.* In Lakhnech, Yassine and Sergio Yovine (editors): *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg, ISBN 978-3-540-30206-3. 8

[23] Boufaied, Chaima, Claudio Menghi, Domenico Bianculli, Lionel Briand, and Yago Isasi Parache: *Trace-checking signal-based temporal properties: A model-driven approach.* In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1004–1015, 2020. 8

[24] *ThEodorE.* `https://github.com/SNTSVV/ThEodorE`, January 2024 [Online]. 10

[25] Koza, John R.: *Genetic programming as a means for programming computers by natural selection.* Statistics and Computing, 4(2):87–112, Jun 1994, ISSN 1573-1375. `https://doi.org/10.1007/BF00175355`. 11

[26] Goldberg, David E.: *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1989, ISBN 0201157675. 11

[27] Gaaloul, Khouloud, Claudio Menghi, Shiva Nejati, Lionel C Briand, and Yago Isasi Parache: *Combining genetic programming and model checking to generate environment assumptions.* IEEE Transactions on Software Engineering, 48(9):3664–3685, 2021. 12, 28

[28] Smith, T.F. and M.S. Waterman: *Identification of common molecular subsequences.* Journal of Molecular Biology, 147(1):195–197, 1981, ISSN 0022-2836. 18

[29] Pearl, Judea: *Statistics and causal inference: A review.* Test, 12:281–345, 2003. 18

[30] Mitchell, Tom M: *Machine learning and data mining.* Communications of the ACM, 42(11):30–36, 1999. 19

[31] Goldberg, David E: *Genetic and evolutionary algorithms come of age.* Communications of the ACM, 37(3):113–120, 1994. 19

[32] Quinlan, J Ross: *C4. 5: programs for machine learning.* Elsevier, 2014. 21

[33] Witten, Ian H and Eibe Frank: *Data mining: practical machine learning tools and techniques with java implementations.* ACM SIGMOD Record, 31(1):76–77, 2002. 21

[34] Hall, Mark, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten: *The weka data mining software: an update.* ACM SIGKDD explorations newsletter, 11(1):10–18, 2009. 21

[35] Witten, Ian H, Eibe Frank, Mark A Hall, Christopher J Pal, and Mining Data: *Practical machine learning tools and techniques.* In *Data mining*, volume 2, pages 403–413, Amsterdam, The Netherlands, 2005. Elsevier. 21

[36] *Appendix: Tool vs Prediction.* `https://doi.org/10.5281/zenodo.12520834`, June 2024 [Online]. 22

[37] Zhao, Xiao Wen, Zhi Hong Guan, Juan Li, Xian He Zhang, and Chao Yang Chen: *Flocking of multi-agent nonholonomic systems with unknown leader dynamics and relative measurements*. International Journal of Robust and Nonlinear Control, 27(17):3685–3702, 2017. 23

[38] Formica, Federico, Mohammad Mahdi Mahboob, Mehrnoosh Askarpour, and Claudio Menghi: *ATheNA-S: a Testing Tool for Simulink Models Driven by Software Requirements and Domain Expertise*. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24),*, New York, NY, USA, 2024. ACM. 23

[39] Reynolds, Craig W.: *Flocks, herds and schools: A distributed behavioral model*. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, page 25–34, New York, NY, USA, 1987. Association for Computing Machinery. 23

[40] Maler, Oded and Dejan Nickovic: *Monitoring temporal properties of continuous signals*. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. 23

[41] Nunez-Letamendia, Laura: *Fitting the control parameters of a genetic algorithm: An application to technical trading systems design*. European journal of operational research, 179(3):847–868, 2007. 25

[42] Poli, Riccardo, William B. Langdon, and Nicholas Freitag McPhee: *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza). 25

[43] Fainekos, Georgios E., Sriram Sankaranarayanan, Koichi Ueda, and Hakan Yazarel: *Verification of automotive control applications using S-TaLiRo*. In *2012 American Control Conference (ACC)*, pages 3567–3572, 2012. 32

[44] Boufaied, Chaima, Claudio Menghi, Domenico Bianculli, Lionel Briand, and Yago Isasi Parache: *Trace-checking signal-based temporal properties: A model-driven approach*. In *International Conference on Automated Software Engineering*, pages 1004–1015. IEEE/ACM, 2020. 33, 35

[45] Menghi, Claudio, Shiva Nejati, Khouloud Gaaloul, and Lionel C Briand: *Generating automated and online test oracles for Simulink models with continuous and uncertain behaviors*. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 27–38, 2019. 33

[46] Diehl, Maximilian and Karinne Ramirez-Amaro: *Why Did I Fail? a Causal-Based Method to Find Explanations for Robot Failures*. IEEE Robotics and Automation Letters, pages 1–8, 2022. 35

[47] Stratan, Cristina, Joshua Dawes, and Domenico Bianculli: *Diagnosing Violations of Time-based Properties Captured in iCFTL*. In *FormaliSE'24: International Conference on Formal Methods in Software Engineering*. ACM, New York, United States-New York, 2024. 35

[48] Chechik, Marsha and Arie Gurfinkel: *A framework for counterexample generation and exploration.* International Journal on Software Tools for Technology Transfer, 9:429–445, 2007. 35

[49] Peled, Doron and Lenore Zuck: *From model checking to a temporal proof.* In *Model Checking Software*, pages 1–14, Berlin, Heidelberg, 2001. Springer. 35

[50] Bernasconi, Anna, Claudio Menghi, Paola Spoletini, Lenore D Zuck, and Carlo Ghezzi: *From model checking to a temporal proof for partial models.* In *Software Engineering and Formal Methods*, SEFM 2017, pages 54–69, Cham, 2017. Springer. 35

[51] Peled, Doron, Amir Pnueli, and Lenore Zuck: *From falsification to verification.* In *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, pages 292–304, Berlin, Heidelberg, 2001. Springer. 35

[52] Mebsout, Alain and Cesare Tinelli: *Proof certificates for SMT-based model checkers for infinite-state systems.* In *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 117–124. IEEE, 2016. 35

[53] Basin, David, Bhargav Nagaraja Bhatt, and Dmitriy Traytel: *Optimal proofs for linear temporal logic on lasso words.* In *Automated Technology for Verification and Analysis*, pages 37–55, Cham, 2018. Springer. 35

[54] Pnueli, Amir and Yonit Kesten: *A deductive proof system for ctl.* In *International Conference on Concurrency Theory*, pages 24–40. Springer, 2002. 35

[55] Balaban, Ittai, Amir Pnueli, and Lenore D Zuck: *Proving the refuted: Symbolic model checkers as proof generators.* Concurrency, Compositionality, and Correctness: Essays in Honor of Willem-Paul de Roever, pages 221–236, 2010. 35

[56] Menghi, Claudio, Alessandro Maria Rizzi, and Anna Bernasconi: *Integrating topological proofs with model checking to instrument iterative design.* In *Fundamental Approaches to Software Engineering*, pages 53–74, 2020. 35

[57] Schuppan, Viktor: *Towards a notion of unsatisfiable and unrealizable cores for ltl.* Science of Computer Programming, 77(7-8):908–939, 2012. 35

[58] Hantry, François and Mohand Said Hacid: *Handling Conflicts in Depth-First-Search for LTL Tableau to Debug Compliance Based Languages.* In *Fifth Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS)*, pages 39–53, Málaga, Spain, 2011. Open Publishing Association. 35

[59] Zheng, Guolong, ThanhVu Nguyen, Simón Gutiérrez Brida, Germán Regis, Marcelo F Frias, Nazareno Aguirre, and Hamid Bagheri: *Flack: Counterexample-guided fault localization for alloy models.* In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 637–648. IEEE, 2021. 35

[60] Bochot, Thomas, Pierre Virelizier, Hélène Waeselynck, and Virginie Wiels: *Paths to property violation: A structural approach for analyzing counter-examples.* In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*, pages 74–83. IEEE, 2010. 35

[61] Griggio, Alberto, Marco Roveri, and Stefano Tonetta: *Certifying proofs for LTL model checking.* In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9. IEEE, 2018. 35

[62] Funke, Florian, Simon Jantsch, and Christel Baier: *Farkas certificates and minimal witnesses for probabilistic reachability constraints.* In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 324–345. Springer, 2020. 35

[63] Timm, Nils, Stefan Gruner, Madoda Nxumalo, and Josua Botha: *Model checking safety and liveness via k-induction and witness refinement with constraint generation.* Science of computer programming, 200:102532, 2020. 35

[64] Gurfinkel, Arie and Marsha Chechik: *Proof-like counter-examples.* In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 160–175. Springer, 2003. 35

[65] Li, Zhong, Yuxuan Zhu, and Matthijs Van Leeuwen: *A survey on explainable anomaly detection.* Transactions on Knowledge Discovery from Data, 18(1):1–54, 2023. 36

[66] Jodat, Baharin A, Abhishek Chandar, Shiva Nejati, and Mehrdad Sabetzadeh: *Test generation strategies for building failure models and explaining spurious failures.* ACM Transactions on Software Engineering and Methodology, 33(4):1–32, 2024. 36

[67] Araujo, João Paulo Costa de, Genaína Nunes Rodrigues, Marc Carwehl, Thomas Vogel, Lars Grunske, Ricardo Caldas, and Patrizio Pelliccione: *Explainability for property violations in cyber-physical systems: An immune-inspired approach.* IEEE Software, 2024. 36

[68] DeMillo, R.A., R.J. Lipton, and F.G. Sayward: *Hints on test data selection: Help for the practicing programmer.* Computer, 11(4):34–41, 1978. 36

[69] Basile, Davide, Maurice H. ter Beek, Sami Lazreg, Maxime Cordy, and Axel Legay: *Static detection of equivalent mutants in real-time model-based mutation testing.* Empirical Software Engineering, 27(7):160, Sep 2022, ISSN 1573-7616. `https://doi.org/10.1007/s10664-022-10149-y`. 36

[70] Do, Van Nho, Quang Vu Nguyen, and Thanh Binh Nguyen: *Evaluating mutation operator and test case effectiveness by means of mutation testing.* In Nguyen, Ngoc Thanh, Suphamit Chittayasothorn, Dusit Niyato, and Bogdan Trawiński (editors): *Intelligent Information and Database Systems*, pages 837–850, Cham, 2021. Springer International Publishing, ISBN 978-3-030-73280-6. 36

[71] Offutt, A. Jefferson and Roland H. Untch: *Mutation 2000: Uniting the Orthogonal*, pages 34–44. Springer US, Boston, MA, 2001, ISBN 978-1-4757-5939-6. `https://doi.org/10.1007/978-1-4757-5939-6_7`. 36

[72] Papadakis, Mike and Nicos Malevris: *An empirical evaluation of the first and second order mutation testing strategies*. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 90–99, 2010. 36