


Università degli Studi di Napoli Federico II

Ph.D. Program in

Information Technology and Electrical Engineering

XXXVII Cycle

Thesis for the Degree of Doctor of Philosophy

Reasoning-based Software Testing

by

Luca Giamattei

Advisor: Prof. Roberto Pietrantuono

Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

"Those who can imagine anything, can create the impossible."

Alan Turing

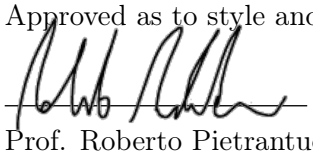
Reasoning-based Software Testing

Ph.D. Thesis presented
for the fulfillment of the Degree of Doctor of Philosophy
in Information Technology and Electrical Engineering
by

Luca Giamattei

October 2024

Approved as to style and content by



Prof. Roberto Pietrantuono, Advisor

Università degli Studi di Napoli Federico II
Ph.D. Program in Information Technology and Electrical Engineering
XXXVII cycle - Chairman: Prof. Stefano Russo

http://itee.dieti.unina.it

Candidate's declaration

I hereby declare that this thesis submitted to obtain the academic degree of Philosophiæ Doctor (Ph.D.) in Information Technology and Electrical Engineering is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.
Parts of this dissertation have been published in international journals and/or conference articles (see list of the author's publications at the end of the thesis).

Napoli, February 5, 2025

Luca Giamattei

Abstract

With software systems becoming increasingly pervasive and autonomous, our ability to test for their quality is severely challenged. Many modern systems operate in uncertain, highly-changing environments, often required to make informed and intelligent decisions autonomously. This results in an intractable state space to explore at testing time. The state-of-the-art techniques try to keep pace, e.g., by augmenting the tester's intuition with some form of (explicit or implicit) learning from observations to search this space efficiently. For instance, they exploit historical data to drive the search (e.g., ML-driven testing) or test execution data itself (e.g., adaptive or search-based testing). Even these data-driven techniques fall short when predicting system behavior in unobserved conditions. Despite clear advances, the need for smarter search in such a vast space keeps pressing.

To overcome current software testing limitations, Reasoning-Based Software Testing (RBST) is proposed, a testing methodology reformulating software testing as causal reasoning tasks. RBST innovates software testing by shifting to a reason-driven paradigm, thanks to Causal Reasoning, contributing to the "Causal Revolution" of Judea Pearl (2011 Turing Award recipient). It constitutes a conceptual leap in how machines and humans cooperate to explore the huge search space and derive tests intelligently. Machines should support and enhance human reasoning far beyond merely identifying patterns in past data. RBST aims to emulate human-like decision-making to "intelligently" navigate the testing space. Leveraging advanced causal discovery and inference techniques, RBST moves beyond traditional ML approaches, enabling more predictive, hypothesis-driven testing. RBST is applied in both stateless and stateful testing scenarios using autonomous driving systems as a case study. Results suggest it has the potential to transform testing practices.

Keywords: Software Testing, Causal Reasoning

Sintesi in lingua italiana

Con i sistemi software sempre più pervasivi e autonomi, garantire la loro qualità diventa una sfida complessa. Molti sistemi moderni operano in ambienti incerti e in continua evoluzione, prendendo spesso decisioni intelligenti in autonomia. Questo porta a spazi di stato estremamente ampi e difficili da esplorare durante il test. Le tecniche di test attuali cercano di affrontare questa sfida migliorando l'intuizione dei tester attraverso l'apprendimento dalle osservazioni, rendendo la ricerca più efficiente. Ad esempio, si utilizzano dati storici (tramite Machine Learning) o dati generati dai test stessi (come nell'adaptive o search-based testing). Tuttavia, anche queste tecniche mostrano limiti nel prevedere il comportamento del sistema in condizioni mai osservate. È necessario rendere la ricerca ancora più "intelligente" per esplorare uno spazio così vasto.

Per superare questi limiti, questa tesi presenta Reasoning-Based Software Testing (RBST), una metodologia che riformula il test del software come attività di ragionamento causale. RBST propone uno shift dai metodi basati sui dati a un paradigma basato sul ragionamento causale, contribuendo alla "Rivoluzione Causale" avviata da Judea Pearl (Premio Turing 2011). Questa trasformazione mira a migliorare la cooperazione tra macchine e umani nell'esplorazione intelligente dello spazio di ricerca, definendo i test in modo più efficace. Le macchine diventano così un supporto attivo al ragionamento umano, superando la semplice individuazione di pattern nei dati passati. Grazie a tecniche avanzate di discovery e inferenza causale, RBST supera i metodi dello stato dell'arte basati su machine learning, generando test più accurati e guidati da ipotesi. RBST è valutato sia in scenari di testing stateless che stateful, con un caso di studio sui sistemi di guida autonoma. I risultati dimostrano che RBST ha il potenziale per trasformare le pratiche di testing.

Parole chiave: Software Testing, Ragionamento Causale

Acknowledgements

I would like to express my sincere gratitude to my advisor, Prof. Roberto Pietrantuono, and my mentor, Prof. Stefano Russo, for their unwavering support throughout my Ph.D. studies and related research. Their patience, motivation, and profound knowledge have been invaluable in guiding me through the research process and the writing of this thesis. I could not have wished for better advisors and mentors during my Ph.D. journey.

A special mention goes to my dear friend and colleague, Antonio Guerriero. I am deeply grateful to have met such a wonderful person. Working and sharing knowledge with him has enriched me both personally and academically, greatly enhancing my Ph.D. experience.

I am also deeply thankful to Prof. Paolo Tonella and his research group for their invaluable supervision and support, particularly during my time abroad at the Università della Svizzera Italiana in Lugano.

My sincere appreciation extends to the thesis evaluators for their insightful feedback and for providing precise and detailed comments, which significantly improved my work.

Finally, I wish to thank the entire Dependable and Secure Software Engineering and Real-Time Systems (DESSERT) research group for fostering a collaborative and welcoming environment. Their contributions made the research laboratory a pleasant and inspiring place to conduct my work and share knowledge.

My work has been partially carried out in the framework of the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 871342 "uDEVOPS".

Contents

List of Acronyms

The following acronyms are used throughout the thesis.

SQA	Software Quality Assurance
AI	Artificial Intelligence
ML	Machine Learning
RQ	Research Question
CR	Causal Reasoning
CI	Causal Inference
CD	Causal Discovery
RL	Reinforcement Learning
CRL	Causal Reinforcement Learning
RBST	Reasoning-Based Software Testing

List of Figures

List of Tables

Chapter 1

Introduction

Motivation Software engineering is an intellectually demanding and creative activity involving complex interdependent tasks aimed at building software products and ensuring their quality. The advancement of Machine Learning (ML) fostered a human-machine co-design view to develop dependable systems [5]: ML algorithms are able to search for significant patterns in large historical datasets gathered throughout the system life cycle, thus supporting several software engineering tasks, aimed for instance at fault avoidance (e.g., testing), fault removal (e.g., debugging) and prediction.

While recognizing patterns in data is a fundamental tool for decision-making, well supported by ML, engineers do much more when building and validating a system. They tend to infer cause-effect relationships among the involved variables, and, based on that, infer hypotheses, simulate possible actions, and derive explanations to then support decisions – in other words, they *reason* on what have learned. Researchers have been trying to explain causality using statistical and ML methods for years. However, these methods are able to identify connections between variables like correlation and regression but fall short in detecting causality. Techniques that merely recognize patterns in data only get us to what Pearl and Mackenzie [141] call the first rung of the causation ladder (i.e., "association"). At this level, we are limited to reason about what has been observed.

Currently, software engineering researchers are exploring the usage of *Causal Reasoning* (CR) methodologies to go beyond a purely data-driven

approach and to exploit the use of causality for finer-tuned strategies. Through the years, researchers have designed a number of techniques able to capture the essence of CR and provide mathematical frameworks for it. This enables the replication of human reasoning on modern machines, greatly enhancing it with computational power. While CR finds consolidated ground in many domains (e.g., epidemiology, economics, social sciences, etc.), it has recently raised interest in software engineering, particularly in the context of software quality assurance (SQA), mainly due to the seminal work of the Turing award-winner Pearl [140].

Problem statement Software testing is a core activity in SQA. It includes a wide range of methods that involve executing the code of the System Under Test (SUT) with a set of inputs and execution conditions, aiming at exposing faults that could potentially lead to system failures [144]. In simple words, software testing is fundamentally about prediction: the primary objective is to determine *what input makes the system fail*. A tester typically envisions how the system will behave under specific inputs or conditions and designs test cases to reveal failures. Traditionally, testers rely on auxiliary information to generate failure-exposing tests, such as using boundary inputs, exploring coverage metrics, or leveraging historical data to predict the most likely points of failure. In recent years, ML has become an integral part of this process: they learn failure patterns and system behaviors from historical data and enable a more efficient and cost-effective testing process through the automation of tasks like test generation, selection, prioritization, and execution.

Despite the indubitable advances, the current testing approaches are limited. They either rely heavily on human intuition to "guess" the most failure-prone inputs or depend on ML to learn from past observations. While human reasoning excels at guessing and hypothesizing potential failure causes, it does not scale when dealing with the vast input space of modern software systems. On the other hand, ML-driven approaches, though powerful, assumes that the future system behavior will resemble the past, which is not always the case, especially in dynamic and constantly evolving systems. This reliance on correlation rather than causation creates a conceptual gap in nowadays testing methodologies: instead of answering the fundamental question, "*What input causes the system to fail?*", they focus on the narrower question of "*What input is more corre-*

lated with failure?" This results in a situation where testing strategies may fail to generalize beyond observed data and may overlook critical failure scenarios. The ability to move beyond mere pattern recognition and search for causality is essential for advancing the field of software testing.

Contribution This thesis supports the claim that a shift from purely data-driven testing methods to a causality-driven paradigm is needed, introducing *Reasoning-Based Software Testing (RBST)* as a methodology to address these limitations. RBST leverages causal reasoning to identify the cause-effect relationships among the variables (e.g., input and output) of a SUT rather than merely recognizing correlations. With techniques from causal inference, RBST enables testers to simulate interventions and counterfactuals - i.e., *What happens if I do/What would have happened if I did* - and explore how different inputs and conditions influence system behavior, net of confounding. Specifically, it allows to actively set inputs and estimate the effect on outcome variables of interest, similarly to how a human tester would do to search the input space. RBST is instantiated and evaluated against state-of-the-art ML-based and search-based testing techniques for both stateless and stateful testing scenarios using autonomous driving systems as a case study. The results show that RBST provides significant improvements in identifying safety-violating test cases.

Ultimately, this thesis proposes that automated causal reasoning is the next frontier in software testing. By adopting a causal reasoning-driven approach, RBST enhances our ability to infer causal knowledge about system behavior, predict failures more accurately, and design more robust testing strategies. This conceptual leap promises to significantly improve software quality assurance practices, ensuring that modern systems are not only tested thoroughly but also understood in terms of their underlying causal dynamics. The rest of the thesis is structured as follows.

- Chapter 2 presents an overview of the evolution of software testing.
- Chapter 3 presents RBST.
- Chapter 4 evaluates RBST for stateless testing.
- Chapter 5 evaluates RBST for stateful testing.
- Chapter 6 summarizes the contributions and outlines future research directions.

Chapter 2

Evolution of Software Testing

2.1 Software quality

2.1.1 Software quality fundamentals

Software quality is defined as the “capability of a software product to conform to requirements" [75, 74]. The goal of Software Quality Assurance (SQA) activities is to establish justified confidence that the software product meets these established requirements [70].

The *SQA process* can be described as a “set of activities that assess both the adherence to and adequacy of the software processes used to develop and modify software products. Additionally, SQA evaluates the extent to which the desired outcomes from software quality control are being achieved" [74].

A system failure occurs when there is a deviation from its expected service. Failures arise due to the presence of *faults* or *defects*, which, when activated, corrupt the system state. If these faults reach the system's interface, they may provoke a failure. As the terminology around software quality can vary across contexts, the definitions used throughout this thesis align with those established by the IFIP Working Group (WG) 10.4 on Dependable Computing and Fault Tolerance. This choice is rooted in the foundational work on dependability by Avizienis, Laprie *et al.* [5], which gives a good overview of the main concepts related to SQA.

Failure A failure is defined as the deviation of a system's delivered service

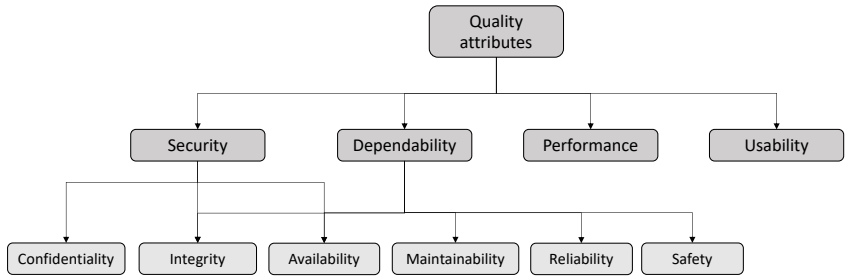


Figure 2.1. Taxonomy of quality attributes.

from its correct service.

Error An error is that portion of the system state which can lead to a failure. A failure happens when the error reaches the service interface, causing the service to deviate from the correct service. [1]

Fault A fault is the adjudged or hypothesized cause of an error. When a fault is *active*, it generates an error; if it is not activated, it is considered *dormant*. Throughout this paper, unless specified otherwise, *fault* refers implicitly to *software faults*. Furthermore, a fault introduced by human error at some point in the software life cycle, which has propagated into the code, is also known as a *software defect* [69] or a *bug*. [2]

2.1.2 Software quality attributes

The SQA process encompasses a range of activities throughout the software life cycle, addressing both *functional* and *non-functional* requirements. To categorize these activities and the associated quality attributes, let us use the taxonomy of dependability and security provided by Avizienis, Laprie, *et al.* [5], extending it to include tasks and attributes beyond dependability and security, as illustrated in Figure 2.1.

[1] In some literature, the term *error* also refers to the human mistake that introduced a fault into the system ([68], [74]).

[2] Note that the IEEE definition of *fault* may differ, as it refers to a defect that has been activated.

For assuring quality, specifically concerning *software* faults, it is possible to consider the following categories:

Fault Avoidance/Prevention This category aims to minimize the introduction of faults into the software. Fault avoidance is typically achieved through good engineering practices, such as effective requirements engineering, design modularity, encapsulation, information hiding, and reuse.

Fault Removal This process seeks to reduce both the number and severity of faults. It includes activities such as fault detection (e.g., testing), fault localization, and the subsequent correction of the code (debugging).

Fault Tolerance Fault tolerance involves techniques that allow the system to tolerate certain failures, usually through mechanisms like error detection and recovery.

(Fault) Forecasting This category involves estimating the current number of faults, predicting future occurrences, and assessing the potential consequences of faults. Additionally, forecasting may include predicting key performance indicators (KPIs) that are not necessarily related to faults but that support quality improvement. This expanded scope is reflected in the parentheses around "Fault," as forecasting here addresses broader quality concerns such as performance and usability [5].

Some of the most common quality attributes that find solid and long-established research communities are:

Dependability attributes. Dependability is the *trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers.* It is an umbrella term encompassing concepts such as reliability, safety, availability and security. We borrow from Avizienis, Laprie *et al.* [5]the following definitions for dependability quality attributes:

- **Availability** refers to the readiness for correct service, defined as "the ability of a system to be in a state to perform a required function

at a given instant of time" (recovery after failure is allowed). It is expressed by the probability of failure-free operation at a given point in time.

- **Reliability** refers to the continuity of a service, that is, for how long it is provided without failures; precisely, it is "the ability of a system to perform a required function under given conditions for a given time interval". Once the system fails, no recovery takes place (while there can be recovery upon component failures). It is expressed by the probability of failure-free operation in a time interval. Note that in the ISO/IEC 25010 [76] standard reliability has a broader meaning, encompassing concepts like availability and maturity.
- **Safety** refers to the absence of catastrophic consequences on the user(s) and the environment; it is the probability of catastrophic-failure-free operation during mission time interval.
- **Maintainability** is the ability to undergo modifications and repairs.
- **Integrity** is the absence of improper system alterations.

Security is a macroattribute encompassing Availability, Integrity and Confidentiality.

- **Confidentiality** is the absence of unauthorized disclosure of information.

This taxonomy provides quantitative (probability-based) definitions of dependability attributes and suits the analysis of the papers regarding dependability and security. For other attributes, the ISO-25010 [76] definitions follow:

Performance is the degree to which (i) the response and processing times and throughput rates, (ii) the amounts and types of resources used, and (iii) the maximum limits of a product or system parameter, when performing its functions, meet requirements.

Usability is the degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use [76].

2.2 Software testing

2.2.1 Software testing fundamentals

Software testing is a core activity within the *fault removal* category of software quality assurance. It includes a wide range of methods that involve executing the code to expose faults that could potentially lead to system failures [144]. Although software testing cannot formally guarantee the complete absence of faults [35], it serves the crucial purpose of demonstrating that the System Under Test (SUT) adheres to its functional and non-functional requirements, thereby providing a valuable indication of its overall quality.

Software testing has the following taxonomy of the main terms [144]:

Test case: A *test case* is a set of inputs, execution conditions, and a pass/fail criterion.

Test case specification: A *test case specification* is a requirement to be satisfied by one or more actual test cases.

Test suite: A *test suite* is a set of test cases. Typically, a method for functional testing is concerned with creating a test suite. A test suite for a program, system, or individual unit may be made up of several test suites for individual modules, subsystems, or features.

Test execution: The *test execution* refers to the activity of executing test cases and evaluating their results. When we refer to "a test," we mean the execution of a single test case, except where the context makes it clear that the reference is to the execution of a whole test suite.

Oracle: An *oracle* is a description of the expected behavior, which is applied to the execution of the program to evaluate, according to the pass/fail criterion, if the observed behavior coincides with the expected one.

With the above definitions, software testing can be defined as "the *dynamic* verification of the behavior of a program on a *finite* set of test cases, suitably *selected* from the usually infinite execution domain, against the

expected behavior" [15]. It is a *dynamic* activity since it always involves the execution of a program under certain inputs (and initial program state or execution conditions). It is considered an infinite activity because, in most cases (even with the simplest programs), a very large number of test cases can be hypothesized, potentially taking years to execute them all. However, this is clearly not viable in practice, where a *finite* number of test cases that fit within the available testing budget is required. Consequently, techniques are needed to select test cases from the vast space of all possible ones. They mainly differ in the *selection* of the "best" test cases, namely those that have a higher probability of exposing an undiscovered fault. Finally, there is a need to evaluate the results of the execution of the selected test cases. Specifically, by building an oracle (e.g., from specifications, requirements, or user expectations), we have a means to decide whether the observed outcomes of program execution are acceptable or not, namely match with the *expected* behavior.

It can be very challenging to homogeneously classify the plethora of activities and techniques employed to test software systems. However, considering the above description as a common ground for all of them, a very effective schema based on **six main questions** has been provided by Bertolino [14]. The schema offers a structured way to explore the key aspects of software testing and to comprehensively address the goals, methods, and scope of the testing efforts.

The first question is *"why" are we testing?* This question refers to the objective, namely to the different properties of the software that we aim to verify. Within this dimension, several objectives can be set, with the most important differentiation being between *functional* and *non-functional testing*, which respectively verify whether the software performs its intended functions correctly and meets its non-functional requirements like performance, reliability, and usability.

The second question is *"how?"* and refers to the strategy employed for the selection of test cases to reach the set objectives. In this regard, many testing techniques have been developed through the years, most of them based on leveraging certain characteristics of the SUT - e.g., *model-based* testing, or *stateful* testing, that aims at exploiting a dynamic description of the possible states of the SUT, like control/data flow graphs - or, more in general, of the inputs that are most likely to cause failures (based on

the tester's intuition, belief, and experience—Section 2.2.2). Other techniques employ methods that have empirically proven to be effective, such as random testing, search-based testing, or, more generally, by applying some algorithmic or statistical technique. Finally, recent machine learning-based techniques leverage observations gathered by monitoring the system in operation and/or collecting historical testing session results to predict the "best" test cases (based on observations—Section 2.2.3). These emerging approaches, driven by data, are particularly useful in large, complex systems where manually determining test cases becomes infeasible.

The third question, *"how much?"*, is answered to set a limit on the number of test cases in a test suite. Several criteria can be adopted, such as reliability—i.e., the probability of failure-free operation for a specified period in a specified environment—or code coverage—i.e., the percentage of code that has been executed. Code coverage can include different levels, such as statement coverage, branch coverage, or path coverage, each providing a different granularity of insight into the testing process. Although many other criteria exist, this question is usually answered within a risk management process, namely by identifying and quantifying the risks derived from remaining (undiscovered) failures. This is especially important in safety-critical domains like aerospace, medical devices, or autonomous vehicles, where the potential consequences of undiscovered bugs can be catastrophic. In such cases, it's essential to balance the cost of additional testing against the risk posed by undetected errors, using methods like failure mode and effects analysis (FMEA) to quantify risks.

The fourth question, *"what?"*, refers to the *level* of the software development and maintenance processes, or in other words, to what parts of the SUT are being tested. For instance, *unit testing* is used to test a single module (e.g., a method or a class in object-oriented programming); *integration testing* is carried out with the objective of finding faults in modules integration and to assess that their interfaces communicate correctly; *system testing*[3] assess the quality of the system as a whole, considering its architectural design.

The fifth question is *"where?"*. The answer is closely related to the previous one and identifies in which environment we test the code. Specif-

[3]Often used interchangeably with *acceptance testing* that evaluate the SUT with respect to requirements and business processes.

ically, whether we test the software in-house, in a simulated environment, or in-field. Testing environments vary significantly depending on the goals. For instance, testing in-house during development is usually done to catch early bugs, while a simulated environment can mimic complex scenarios or external dependencies. On the other hand, field testing, sometimes referred to as production testing, can be performed online, in the production environment of the actual system, or offline, in the production environment but with a SUT separated from the actual system. It offers feedback on real-world performance but also carries higher risks, especially when performed online, in case failures affect actual users. However, techniques like A/B testing or canary releases allow for testing in production in a controlled way, by gradually rolling out new features to a subset of users before full deployment [10].

The sixth question, *"when?"*, defines the phase in the software lifecycle in which we are performing testing. Testing is typically conducted at various stages: during development, at the end of development (i.e., before release), and after deployment (i.e., in the maintenance phase). Early testing during development can help identify and resolve issues earlier in the lifecycle, reducing the cost and effort of fixing bugs later. Continuous testing, which integrates testing into the CI/CD (Continuous Integration/-Continuous Deployment) pipeline, ensures that testing is ongoing as code changes are made. Post-deployment testing, such as *regression testing*, ensures that updates, patches, or new features do not break existing functionality or degrade system performance.

This thesis proposes a novel methodology to answer the second question (i.e., "*how?*" we generate test cases).

2.2.2 Software testing guided by intuition and belief

Answering the "*how?*" question is ultimately a *prediction* task. When deriving test cases, a tester essentially asks themselves: *What input would make the system fail?* They try to anticipate how the SUT would behave for specific input in a given execution scenario. In some cases, their skill, intuition, and experience with similar systems are used to prepare a few, specific test cases that are expected to expose the system's issues. In other cases, the tester is required to dynamically design and execute test cases while interacting with the system, leveraging their experience to update

the test cases. These methods are usually referred to as "ad hoc" and "exploratory" testing, respectively. While they may seem straightforward, they are intellectually demanding, costly, and error-prone activities that involve manual steps [38].

Automation is key to enabling the development of effective and efficient software testing techniques [61, 14]. To enable automation and, at the same time, bring rigor to the testing process, testers typically rely on some form of auxiliary information. This information supports the tester's predictions in deriving failure-exposing tests and enables the design of automated testing techniques that exploit the tester's belief about what information is expected to correlate with failures. This conceptually shifts and complements the question *"What input would make the system fail?"* to *"What proxy information can I use to guess failure-causing input?"*. In other words, the "how" answer is translated into a guessing problem, where testers are required to identify a failure-related auxiliary variable to use.

S/he may use the software specification and design what are called **specification-based** techniques (often called *black-box* as they need only the SUT specification, without taking into account its internal structure). Here the belief is that some sort of information can be extracted from the specification to drive the input generation. An example is "equivalence partitioning", where the input domain defined in the specification is subdivided into a set of equivalence classes (according to a certain relation - e.g., range or type of the input) from each of which a representative set of test cases is taken. The belief here is that maximizing the portion of the input domain explored is expected to increase the probability of finding issues and the confidence in the correct functioning of the system under expected inputs. This technique is opposed to the "boundary-value analysis" and the extension called "robustness testing" in which, respectively, inputs are chosen near and out of the domain boundaries, under the belief that many faults concentrate near the extreme values of inputs. Another specification-based technique is "random testing", where inputs are selected randomly within the input domain, without focusing on specific boundaries or categories. The goal is to explore a wide range of input values in an unbiased manner, which can sometimes reveal unexpected bugs that targeted techniques might miss. Although random testing offers simplicity and can be easily automated, it lacks the precision of other methods

like equivalence partitioning or boundary-value analysis. It is particularly useful for finding errors in large or complex input spaces, but may not be as efficient in detecting faults concentrated around boundary conditions or specific equivalence classes. Other categories include but are not limited to, model-based, code-based, fault-based, and usage-based testing techniques.

Model-based testing consists of extracting a behavioral model (e.g., a sequence of input state transitions, as in finite state machines) of the SUT, from specification, design, or code, and using it to produce test case specifications that can reveal discrepancies between actual program behavior and the model [144].

Similarly, **Code-based** techniques (often called *white-box*) rely on the assumption that higher coverage of the exercised SUT structures (e.g., paths, instructions, and branches) leads to greater confidence in the absence of faults. From this intuition, various sub-categories have been developed, each with a different level of granularity. For example, testers may utilize a model-based approach and use information from the control-flow or data-flow to compute coverage metrics. In the case of control-flow, test requirements are based on the execution flow of the SUT. Statement testing, branch testing, and condition/decision testing are examples of techniques. They measure the coverage as a percentage so that, in the example case of branch testing, when all branches have been executed at least once by the test cases, 100% branch coverage is said to have been achieved [15]. With the data-flow category, major focus is put on variables definition and usage within a program. A number of coverage criteria can be defined, like the coverage of the paths of variables from definition to usage. Worth noting that code- and specification-based techniques, often compared as *structural* and *functional* testing, are not necessarily alternatives but rather complementary, depending on the testing budget and code availability.

Fault-based techniques design test cases specifically aimed at exposing likely or predefined categories of faults, based on the belief that discovering simple syntactic faults can lead to finding more complex, real faults [15]. One of the most popular techniques in this category is *mutation testing*, which involves creating "mutants", or slightly modified versions of the SUT with pre-defined faults (i.e., "mutation operators"). Test cases are

executed on both the original SUT and the mutants, and the differences between the results are analyzed. If a test case produces a different result for a mutant, the mutant is considered "killed." The goal of mutation testing is to generate test cases that can "kill" all surviving mutants or, in other words, uncover all seeded faults.

Finally, **usage-based** techniques, such as *operational testing*, define inputs based on a probability distribution or profile that reflects their likelihood of occurrence in the actual operation of the SUT. The assumption behind this approach is that to maximize confidence in testing for reliability, the test environment should closely mimic the operational environment. By testing the system with inputs that match real-world usage patterns, these techniques aim to identify failures that are more likely to occur in practice, thereby improving the system's reliability and robustness in real-world conditions.

The mentioned techniques have proven to be highly effective in many domains and with various SUT types, but they may not scale well with modern software systems, which have become increasingly complex and autonomous. On the other hand, due to the vast amount of data produced by software systems (e.g., from monitoring tools and testing sessions), a new wave of data-driven testing techniques has been proposed, leveraging Artificial Intelligence (AI) and particularly Machine Learning (ML) [38].

2.2.3 Software testing guided by observations

AI techniques have been successfully employed to reduce the effort required for numerous software engineering activities [195, 38, 191]. Among these techniques, ML, a research field that bridges AI, computer science, and statistics, emerges as one of the most used to automate these activities. Specifically, many challenges in software testing can be formulated as learning problems and addressed through ML algorithms. Consequently, there is increasing interest in leveraging ML to automate and enhance the efficiency of software testing.

As previously mentioned, one key aspect of software testing involves accurately predicting the *output* that a system will produce for a given *input*. This ability would enable testers to identify the most effective test cases, which helps reduce testing costs and improve the process of test case generation, selection, and prioritization. Automating this prediction process

is essential for developing scalable software testing techniques, especially as software systems become more complex. Automation has been facilitated by the growing availability of data on software systems, coupled with advancements in ML. These developments have fostered a human-machine co-design approach, in which both humans and machines work together to develop strategies that can detect significant patterns in large historical datasets gathered throughout the system's life cycle. These strategies support several software engineering tasks, including software testing.

Researchers and practitioners are increasingly exploring ML-based strategies that involve learning from observational data to support the aforementioned prediction. The underlying assumption is that meaningful information can be extracted from observational data to derive effective test cases. As noted by Durelli et al. [38], learning from such data enables the automation of critical testing tasks.

In these learning-supported strategies, the tester's belief about which inputs might cause system failures is complemented by historical observations. These observations are typically employed in one of two ways: *i)* they are used to train an ML model used to predict a certain output of interest for the testing task, like to predict failure-causing inputs based on learned patterns [6, 38, 135]; or by adaptively using them to exploit feedback from test executions and improve subsequent tests(e.g., adaptive testing [127] and search-based software testing [118, 59]).

In general, ML-based strategies are designed to automatically learn patterns from the input data. This means determining which features are most *correlated* with failures. This means further refining the question *"What proxy information can I use to guess failure-causing inputs?"*, into *"What information is most correlated with failures?"*

ML is widely used in both traditional and AI/ML-based systems, as it naturally lends itself to a data-driven approach. Supervised and unsupervised ML techniques are applied to test generation, selection, and prioritization [38, 17, 147, 135, 73, 180]. These techniques find applications also in Learning-Enabled Autonomous Systems (e.g., autonomous driving cars), where traditional pre-release testing is often impractical, *field testing with operational data* becomes essential. In this context, ML plays a pivotal role, enabling continuous system improvement through data-driven feedback loops. For instance, decision trees [13] and autoencoders [172] have

been used, but also Reinforcement Learning (RL) is being explored, especially for implementing test agents in GUI application testing [2, 42, 117], including Android apps [154] and Windows OS [60], as well as for *test selection and prioritization* [165]. Though the potential of RL remains largely untapped, recent work highlights a growing interest in RL for test generation [7, 56]. The use of RL for test generation, often known as *stateful* testing, usually consists in modeling the testing problem as a markov decision process, where sequences of inputs have to be selected, while observing the SUT state transitions. In general, the emerging AIOps trend refers to the application of ML algorithms to drive quality assurance activities, such as log management, code completion [101, 30, 21], and more. In parallel, the MLOps trend emphasizes the use of field data to improve ML models within systems through retraining [64].

Another significant approach within the domain of learning-based testing involves leveraging state-machine inference to derive models from sequential observations. This line of work inherently incorporates causality, as it accounts for how outputs are generated by specific input sequences. One notable reference in this area is the tutorial by Meinke *et al.* [119], which provides a comprehensive overview of learning-based testing. The authors illustrate how learning algorithms and model checkers can collaborate in a feedback loop to iteratively refine system models, optimizing test case generation. This paradigm demonstrates how causal dependencies in system behaviors can enhance automated, specification-based black-box testing by improving test case relevance and efficiency.

In summary, these data-driven testing techniques—closely related to the work in this thesis—have enabled a high degree of automation in software testing. They have fundamentally shifted the focus from designing individual test cases to framing testing challenges as automated prediction problems. However, they still present several limitations, which are discussed in the next section, that this thesis aims to address.

2.3 Limitations of current software testing mindset

The current approach to testing has inherent limitations. It either relies only on human intuition (expertise/experience) to “guess” the right

failure-correlated information, and then searching for the failure-exposing (test) cases from this information; or it is at most supported by ML models to learn this correlation from past observations (e.g., both for testing/QA like in AIOps, and for evolution, like in MLOps). Human reasoning is, in principle, great at this guessing task, but it does not scale, especially with the huge state space of modern complex and autonomous software systems: it cannot systematically explore what auxiliary information is better correlated with failures, and even if this information is given, cannot efficiently draw test cases from it. ML can help reduce the search space, but it is just a palliative: ML is a learning-from-association paradigm [141], in which models learn patterns on observed occurrences in a certain context, and make "predictions" based on what seen – it is said to be a "passive" prediction approach and hardly generalize beyond what seen. In essence, ML needs to wait for relevant knowledge to "come out" by passively looking at what is happening [140]. In other words, learning correlation rather that actual causation, inherently assume that the future context resembles the past, which may be not the case in constantly evolving software systems. This way, there is no possibility to implement a scalable quality-by-design approach (wherein the potentially dangerous scenarios are automatically foreseen during design/evolution before release); rather, ML is limited to at most a tolerance-for-poor-quality strategy (e.g., trigger an action when some deviation from the past is detected a dangerous behaviour is suspected).

By looking at the presented evolution of software testing, clearly it has experienced a substantial shift in the mindset. This shift hides a conceptual glitch (summarized in Figure 2.2): with current learning-based techniques such as ML-driven testing, we are implicitly turning the *What input makes the system fail?* question into *What information is more correlated to failure?* Clearly, this is not the kind of prediction a tester tries to do: with the first question, s/he means: *What input causes the system to fail?* The question based on correlation reads as: *Given the same context in which I learned the model (i.e., the same data distribution), what output Y do we expect if the input X happens to be equal to x?* On this basis, for instance, we select or prioritize tests that are more similar to failure-causing tests observed in the past, and ML is great at this task. On the other hand, the causal question reads as: *What output Y do we expect if the input*

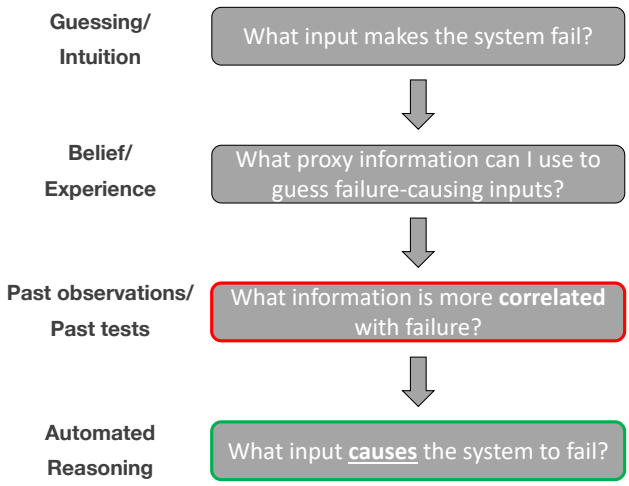


Figure 2.2. Evolution of testing mindset and proposed shift.

X *is actively set to* x (hence, if we change X's distribution)? The tester's question is inherently *causal*, but a learning-from-association paradigm like ML cannot handle causality [140].

Besides the mentioned limitations, ML-driven approaches are exposed to *confounding bias*. For instance, assume the activity of testing an Autonomous Driving System (ADS) with the objective of finding violations of the requirement by which the ADS (e.g., an autonomous driving car) must stay in its lane. Consider a simplified example with two input variables: *Road Type* R (e.g., straight road, right turn, left turn) and *Vehicle Target Speed* V (e.g., the speed of the vehicle in km/h). The requirement is described by one variable *Distance from Center of the lane* D (e.g., distance in meters from the center). The tester objective is to find the combination of inputs for which D exceeds a threshold, indicating that the vehicle went out of its lane. Assume s/he has a set of observations of ADS driving and, as shown with the arrows in Figure 2.3, that there is a correlation between V and D but also between R with both V and D.[4]

[4]It is reasonable to assume that from observations of a vehicle in operation at high speed, there is an higher chance to go out of lane. Similarly, at a turn we can assume that the speed would be lower with respect to a straight road and there is a higher probability of exceeding the lane boundaries.

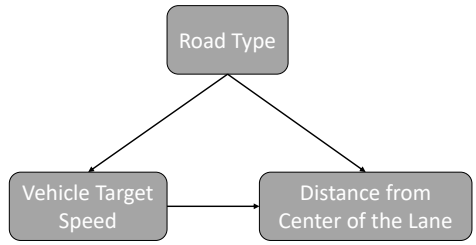


Figure 2.3. Confounding example.

In this case, R is said to be a *confounder*: in fact, if we observe a correlation between V and D this may be well due to the influence of R on both, and we may falsely conclude that the V causes D. The consequence is that if tester generated test cases based just on this observed correlation $P(D|V)$, s/he might fail to expose violations of the distance from the lane center requirement by just changing the vehicle target speed and ignoring the road type. This is also known as the *Simpson Paradox*: "Any statistical relationship between two variables may be reversed by including additional factors in the analysis" [140]. In the above example, it would mean that by solely increasing V does not increase D without considering R. As detailed in the next chapter, randomized controlled trials - often feasible in software testing - naturally account for the confounding bias. However, reasoning techniques solve this issue more efficiently and without requirements on data collection - i.e., even in the presence of *observational* data like operational data gathered through system monitoring.

This thesis proposes a further shift in the conceptual approach to software testing, with the use of causal reasoning. It aims to go beyond the mentioned limitations of current testing techniques, by *i)* providing support to answer the (correct) question "*What input causes the system to fail?*" and *ii)* employ causal reasoning techniques to learn the generative process of observations, and to account and adjust for confounding bias and to make more accurate predictions.

Chapter 3

Reasoning-based Software Testing (RBST)

3.1 Causal Reasoning (CR)

CR has its roots in the 20^{th} century. It had its first applications in a variety of domains, like epidemiology [63], economics [71], social and medical sciences [126, 116], education [29, 92], with significant impact on better understanding and explaining complex phenomena, making predictions, and managing decision processes. CR mimic something natural in human logic, namely the identification of causes and effects and the answering of "*what if*" (causal) questions. Pearl and Mackenzie [141] have conceptualized a framework describing three levels of understanding and reasoning about causation, called the "ladder of causation". The first level is association (correlation), based only on what we observe. It is a basic understanding that two variables tend to co-occur, making no claims about causation. CR allows to stair up to the second and third rung of the ladder, by performing, respectively, "interventions" (evaluating the impact of actively manipulating one variable to observe the effect on another) and "counterfactuals" (exploring hypothetical scenarios, that did not occur but help understand what might have happened under different conditions).

Through the years a number of frameworks have been proposed to allow machines to reason with causality, by modeling cause-effect relations and quantifying the effects of causes through what is called "causal inference".

3.1.1 Causal frameworks

Causal frameworks are mathematical and/or graphical representations of causal relationships within an individual system or population [132, 65]. They embed some form of external knowledge, like prior knowledge of the data generation mechanism and assumptions about plausible causal mechanisms, that distinguishes them from associational methods [82]. They are used to learn and estimate causal effects between involved variables of a certain system, facilitating causal inference. One of the first conceptual frameworks is the "potential outcome" by Neyman [168, 129] and Rubin [157] (also called Neyman-Rubin Causal Model). It allows estimating counterfactuals, i.e., estimating the causal effect of a treatment on an individual by comparing what happened (*observed outcome*) to what would have happened in the absence of that treatment (*potential outcome*). The formal definitions follow [140, 132, 53].

Definition 1 ***Potential Outcome (PO)***
A potential outcome $Y_{T=t}(u)$ of a variable Y is the value that Y would have taken for individual u, had T (treatment variable) assumed the value t.

A potential outcome is fundamentally distinct from an actual outcome. The first is a hypothetical outcome that, differently from the second, has not been observed. For instance, in case of a binary treatment $T = \{0, 1\}$, if we observe an outcome $Y_{T=1}(u)$, we cannot observe $Y_{T=0}(u)$ at the same time – e.g., if an individual took a medicine ($T = 1$), we cannot observe the case in which (s)he did not take it ($T = 0$)). This is a potential outcome (also called *counterfactual outcome*), and one retrospectively reasons about *"what would have happened if"*, as opposed to the observed outcome (or *factual outcome*).

The PO framework, defined at the level of an individual, aims to estimate a potential outcome and then compute the treatment effect. An example metric is the *Individual Treatment Effect* (ITE):

$$ITE(u) = Y_{T=1}(u) - Y_{T=0}(u) \tag{3.1}$$

The impossibility of simultaneously observing both potential outcomes, and consequently observing the effect of a treatment t on an individual u, is the *fundamental problem of causal inference* [66]. However, statistical

solutions allow to replace this impossible problem of observing the causal effect of t on a specific individual with the feasible estimation of the average causal effect of t over a population of individuals. At the population level, we define the *treated* group, including all the treated individuals ($T = 1$), and the *control* group ($T = 0$). For simplicity, we use a binary treatment; otherwise multiple treated groups are formed. The ITE computed at population level is called *Average Treatment Effect* (ATE):

$$ATE = E[Y_{T=1} - Y_{T=0}] \tag{3.2}$$

ATE is also often called Average Causal Effect (ACE) in the counterfactual literature [126]. In general, ACE refers to the estimation of the effect of the change of value of a certain cause to a variable of interest [140]. More broadly average causal effects include a variety of metrics, such as Average Treatment effect on the Treated group (ATT), and Conditional Average Treatment Effect (CATE) [132].

To compute (when feasible) an unbiased estimation of these metrics, randomized controlled trials (i.e., random assignment of *control* and *treatment* groups) are performed. Otherwise, in presence of observational data, a number of causal inference methods can be employed (introduced in the next section). To ease the estimation of treatment effects, some assumptions are usually made, such as the Stable Unit Treatment Value Assumption (SUTVA) [28, 158]. SUTVA specifies that the value of Y for unit u when exposed to treatment t will be the same no matter what mechanism is used to assign treatment t to unit u and no matter what treatments the other units receive.

Another powerful way of dealing with causality is to use graphical models. For instance, the Bayesian Network is a graphical representation of probabilistic relations among variables. It is characterized by a Direct Acyclic Graph (DAG), where nodes represent the variables and edges represent conditional dependencies among variables [132].

Definition 2 ***Bayesian Network (BN)***
A BN is a pair $\mathscr{B} = (\mathscr{G}, P)$ where P factorizes over the graph $\mathscr{G}$ and is expressed as a product of conditional probability distributions associated with $\mathscr{G}$'s nodes $P(X_1, ..., X_n) = \prod_{i=1}^{n} P(X_i|Pa^{\mathscr{G}}_{X_i})$ where X_i are the nodes of $\mathscr{G}$ and $Pa^{\mathscr{G}}_{X_i}$ are their parents, namely the nodes with directed edges

pointing to X_i.

BN embeds probabilistic knowledge, not necessarily causal knowledge. Indeed, *BN* are primarily for estimating the likelihood of one event occurring based on the observation of another (first level of the Pearl's ladder of causation).

In causal inference we are interested in the distribution of an outcome variable X_i after *actively setting* another variable X_k to a certain value x (i.e., *doing* an intervention, applying a treatment). Pearl introduced the *do-operator*, a mathematical representation of physical intervention, written as $P(X_i|do(X_k = x))$ [140, 139]. The *do*-calculus, along with estimation methods [132], supports the inference of type $P(X_i|do(X_k = x))$. An intervention $do(X_k = x)$ changes the DAG (hence the distribution), by removing the causal relations with its predecessors (i.e., deleting the $Pa(X_k) \rightarrow X_k$ edges), meaning that X_k is no longer affected by any other variable. A *do*-intervention changes the data generative process, thus: $P(X_i|do(X_k = x)) \neq P(X_i|(X_k = x))$. To account for *do*-interventions, and to go over the first level of the Pearl's ladder of causation, a BN can be interpreted as a *Causal BN* and the factorization formula in Definition 2 extended to give an interpretation of an intervention with a *truncated factorization* : $P(X_i|do(X_k = x)) = \prod_{i \neq k} P(X_i|Pa^{\mathscr{G}}_{X_i})$.

In any *BN*, the edges into X_i mean that the probability of X_i is governed by the conditional probability tables for X_i, given observations of its parent variables. In *Causal BN*, the conditional probability tables specify the probability of X_i given interventions on the parent variables [140, 141]. A *Causal BN* can also be defined as a Causal Direct Acyclic Graph where nodes are variables and edges represent cause-effect relationships through conditional probabilities. In general, a Causal Direct Acyclic Graph can be defined as follows:

Definition 3 ***Causal Direct Acyclic Graph (CausalDAG)***
A CausalDAG $\mathscr{G} = (\boldsymbol{X}, \mathscr{E})$ *is a directed acyclic graph that describes the causal effects between variables, where* $\boldsymbol{X}$ *is the node set and* $\mathscr{E}$ *the edge set. In a CausalDAG, each node represents a random variable including the treatment, the outcome, and other observed and unobserved variables. A directed edge* $x \rightarrow y$ *denotes a causal effect of* x *on* y.

Another important use case of a CausalDAG is the Functional Causal Model (FCM). In FCMs the value of each variable X_i is assumed to be a deterministic function of its parents $Pa(X_i)$ and of the unmeasured disturbance U_i ($X_i = f(Pa(X_i), U_i)$) - the same idea of representing stochasticity as in Variational Auto-Encoders with the *reparametrization trick* [84]. These are a non-linear non-parametric generalization of linear Structural Equation Models (SEMs). A CausalDAG using an FCM for conditional distributions is called a Structural Causal Model (SCM) [140].

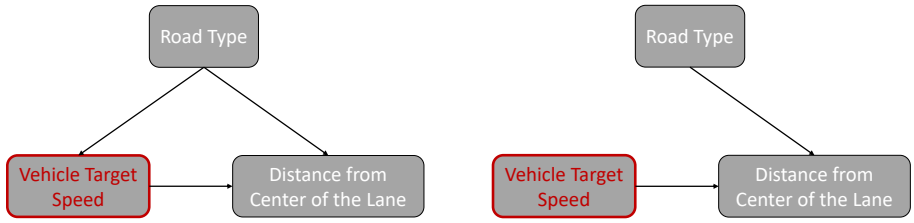
Definition 4 ***Structural Causal Model (SCM)***
A Structural Causal Model is a causalDAG $\mathcal{G} = (\boldsymbol{X}, \mathscr{E})$ where causal relationships $\mathscr{E}$ are described as a collection of structural assignments $X_i := f_i(Pa(X_i), U_i)$ that define the (endogenous) random variables X_i as a function of their parents $Pa(X_i)$ and of (exogenous) independent random noise variables U_i.

An SCM enables the estimation of the effects of both interventions and counterfactuals. Although modeling frameworks based on PO and SCM are logically equivalent when addressing counterfactual questions [140], they differ in their underlying assumptions and treatment of counterfactuals. In the PO framework, counterfactuals are treated as undefined primitives. In contrast, SCMs derive counterfactuals from more fundamental concepts, such as causal mechanisms and their structure [141].

In an SCM, there are no conditional probability tables as in *BN*, the edges simply mean that X_i is a function of its parents, as well as the exogenous variable U_{X_i} [141].

Figure 3.1a shows an example of an SCM from the example in Section 2.3, where *vehicle target speed* causally affects the distance from lane center, and *road type* affects both speed and lane center distance. An intervention $do(X_k = x)$ changes the SCM graph (hence the distribution), by removing the causal relations with its predecessors (i.e., deleting the $Pa(X_k) \rightarrow X_k$ arrows, see Figure 3.1b), meaning that X_k is no longer affected by any other variable. Interventions are modeled as follows [143].

Definition 5 ***Intervention distribution.*** *The probability $P(X_i|do(X_k = x))$ over an SCM is the distribution entailed by the SCM obtained by replacing the definition $X_k := f_k(Pa(X_k), U_k)$ with $X_k := x$.*



(a) Original SCM

(b) SCM after an intervention on *Vehicle target speed*

Figure 3.1. Example of SCM and intervention

The confounding problem shown in Section 2.3 (also called *back-door pattern*) is hereby solved by asking for the causal effect by doing an intervention (obtaining the graph in Figure 3.1b), namely $P(D|do(V))$. By doing so we obtain the real effect of V on D, net of R. Indeed, by the *do*-calculus rules, we come to marginalizing over R to get the desired effect: $P(D|do(V = v)) = \sum_r P(D|V = v, R = r) \cdot P(R = r)$. Note that back-door is the simplest case: more complex patterns can entail far longer chains of transformations that are not easy to derive without *do*-calculus. Its rules not only dramatically simplify the transformations, but have also been shown to be *complete* [67]: if a causal effect is identifiable, there exists a sequence of applications of the three rules that transforms the causal "*do*" formula into a formula containing only observational quantities (i.e., conditional probabilities).

A number of other models and frameworks to deal with causality have been conceptualized through the years. Difference-in-Differences (DiD) [94], a common framework in econometrics and health care for estimating the ATE, compares the changes in outcomes over time between a treatment group and a control group [132]. DiD is based on typical assumptions made also for the PO framework, like SUTVA, but requires additional key assumptions. For instance, it assumes that in the absence of treatment, the average treatment and control groups would have followed parallel trends and that in the pre-treatment period the treatment had no effect on the pre-treatment population [94]. Fuzzy Cognitive Maps [89] are graph structures for representing causal relationships, with their strength encoded

as fuzzy values. It is especially applicable to soft knowledge domains, in presence of uncertain system concepts and relationships. It implements a fuzzy causal algebra aiming at estimating, through backward and forward chaining, *direct* and *indirect* effects between variables [89]. However, differently from previous frameworks, its purpose is not to provide rigorous and precise causal methods, but "semi-quantitative" (i.e. using and producing indicative rather than predictive numerical values) [8]. Finally, Causal trees [4] are types of decision tree structure that incorporate causal modeling principles. They are based on the PO framework and estimate treatment effects (usually Heterogeneous Treatment Effect [185]) with the use of tree-based methods, like classification and regression trees [4, 181].

3.1.2 Causal Inference (CI)

CI aims to estimate the causal effect of a specific variable (referred to as the treatment) on an outcome of interest [132]. There are two primary strategies for quantifying causal effects: performing randomized experiments, where treatment is randomly assigned, or using CI methods to estimate the treatment effect from observational data. Randomized experiments are the gold standard, they enable causal identification by design but can be too costly, unethical or otherwise not feasible for certain situations [197]. In contrast, methods that rely on observational data have gained popularity due to the abundance of available data.

Given a causal framework, the fundamental objective of CI is to estimate how the outcome variable Y changes when we intervene to assign a specific value t to the treatment variable T (i.e., $P(Y|do(T = t))$), net of confounding.

Reaching this objective involves several steps. The first step is the *identification*, which consists of distinguishing spurious correlations from true causal effects and determining whether and how a causal effect can be estimated from available data. It consists in finding the mathematical formula that generates the answer to a causal query or, in other words, a "statistical quantity to be estimated from the data that, once estimated, can legitimately represent the answer to the query" [141]. However, it may not always be possible to identify this quantity: for example, imagine an unmeasured variable Z ("Road Type" in Figure 2.3) that affects both X and Y (respectively "Vehicle Target Speed" and "Distance from Center of

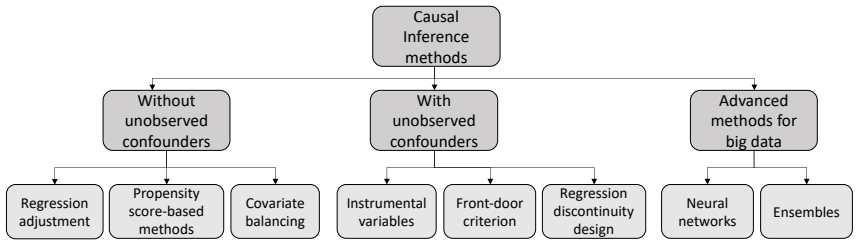


Figure 3.2. Taxonomy of Causal Inference methods [53].

the Lane") with X also having a direct effect on Y. In this case, identifying the causal effect $P(Y|do(X))$ is challenging due to the confounding introduced by Z. To address this, model refinement or additional assumptions (e.g., assuming Z's effect is negligible) may be necessary. In graphical causal models, the *do*-calculus provides a set of rules that leverage the structure of the causal graph to transform an interventional/counterfactual query into an observational query. In general, a number of methods can be exploited to control for *confounding bias* (e.g., the *back-door*, *front-door*, *mediators, instrumental variables* [141, 140, 72]). The second step is the *estimation* of a causal effect from observational data. Once identification is established, a variety of statistical methods can be used for this purpose [132], including propensity-based stratification [156], propensity score matching [18], inverse propensity weighting [184], regression discontinuity [177], two-stage least square [176], and generalized linear models [132]. For our investigation, we consider the classification given by Guo *et al.* [53] on methods to estimate causal effects starting from observational data[1], shown in Figure 3.2. Depending on the assumptions of data, CI methods are separated into two categories: observational data with unconfoundness and with confounding (i.e, without and with unobserved confounders).

Methods without unobserved confounders include three categories of *adjustment*: *regression adjustment*, *propensity score-based methods*, and *covariate balancing*. Adjustments eliminate the confounding bias based on a set of observed features. In essence, these methods isolate the true

[1]In absence or impossibility to gain experimental data (e.g., randomized controlled trials)

causal effect of the treatment by statistically controlling the influence of confounding variables, assuming that all confounder variables are among the observed ones.

Methods with unobserved confounders relax the assumption of unconfoundness, making them more related to real-world scenarios where unobserved confounders may play a role. In terms of SCM, the assumption of unconfoundness implies that conditioning on a subset of observed variables blocks all back-door paths. These are indirect paths between the treatment and outcome through confounders, which, if not blocked, introduce bias into the causal effect estimation. Without the unconfoundness assumption, conditioning is insufficient, and alternative information must be utilized. These methods include *instrumental variables*, *front-door criterion*, and *regression discontinuity design*.

With the increasing availability of Big Data, machine learning-based methods have been designed for learning causal effects, mainly through the use of neural networks and ensembles. To go beyond the unconfoundeness assumption, most of these methods learn representations of confounders, under the assumption that they can be learned from observational data [53]. For instance, Louizos *et al.* [106] propose a Causal Effect Variational Autoencoder (CEVAE), aiming at estimating a latent-variable model where they simultaneously discover the hidden confounders and infer how they affect treatment and outcome.

Finally, following the Pearlian inference methodology, recently Blöbaum *et al.* [16] have proposed a *simulation-based* method called *Do-Sampler*, based on sampling from post-intervention distribution of variables. Given a set of interventional variables, *Do-Sampler*: *i)* cuts all their incoming edges (they are intervened, hence no longer causally affected by any other), *ii)* sets the values of these variables to their interventional quantities, and *iii)* propagates those values through the causal graph to compute interventional outcomes with a sampling procedure. Comprehensive discussions on CI methods have been published by Guo *et al.* [53], Yao *et al.* [197], and Nogueira *et al.* [132].

3.1.3 Causal Discovery (CD)

CD aims at learning causal relationships between variables [132] to build a causal framework explicitly embedding cause-effect relationships,

i.e., graphical models.[2] These can be built in three main ways: by randomized controlled experiments[3], manually with domain knowledge, or by employing CD algorithms on observational data [49]. Recently attention has been put on the latter, which aims at extracting a graphical causal structure from observational data, avoiding controlled experiments, which may be too expensive or even technically infeasible [164]. To achieve this, CD algorithms assume that causality can be derived from statistical dependencies. These algorithms rely (to a different extent) on various subsets of assumptions, the main ones being the *Causal Markov*, *Faithfulness*, and *Sufficiency* assumptions [40, 49, 132, 187], to derive correspondences between the (conditional) independence in the probability distribution and the causal connectivity relationships in the generated DAG [40]. The first two conditions are the most important ones and respectively state that: *i)* every vertex X in the graph G is probabilistically independent of its non-descendants given its parents; *ii)* if a variable X is independent of Y given a conditioning set Z in the probability distribution (i.e., X and Y are independent conditional on a set of variables Z if knowledge about X gives no extra information about Y once you have knowledge of Z) then X is d-separated from Y given Z in the DAG (in other words, the statistical dependence between variables estimated from the data does not violate the independence defined by any causal graph that generates the data [53]). To understand d-separation, let X,Y, and Z be disjoint subsets of all the vertex in the DAG; then Z *d-separates* X and Y just in case every path from a variable in X to a variable in Y contains at least one vertex X_i such that either: *i*) X_i is a *collider* (i.e. the arrows converge on X_i in the path), and no descendant of X_i (including X_i) is in Z; or *ii*) X_i is not a collider, and X_i is in Z [65]. Markov and faithfulness conditions are sufficient to define an equivalence structure over directed acyclic graphs, where graphs that are in the same Markov equivalence class have the same (conditional) independence structure. Sufficiency requires that, for a pair of observed variables, all their common causes must also be observed in the data.

Wang *et al.* [187] provide a classification of CD methods, shown in Figure 3.3. In particular, they are divided in *contstraint-based*, *score-based*,

[2]Non-graphical causal frameworks, such as PO, do not require CD.

[3]This can also be done with *soft* interventions, which influence the intervened variables distribution without setting it to a fixed value [41, 86].

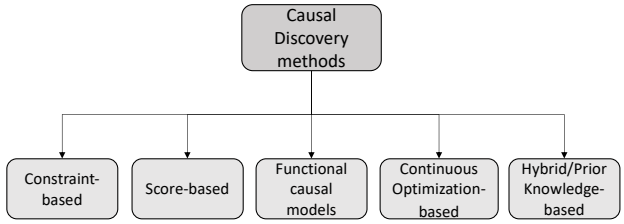


Figure 3.3. Taxonomy of Causal Discovery methods [187].

FCM-based (Functional Causal Models), Continuous Optimization-based, and Hybrid or Prior Knowledge-based methods.

Constraint-based algorithms use conditional independence tests on observed data to identify a set of edge constraints [166]. These algorithms have the advantage of being generally applicable, despite being based on the strong assumption of causal faithfulness, therefore often requiring large sample sizes to perform well [49]. Typical (conditional independence) constraint-based algorithms are PC, FCI [166] and its improvement RFCI [25].

Score-based algorithms optimize a score assigned to candidate graphs, exploiting adjustment techniques such as the Bayesian Information Criterion, that approximates the posterior probability of the model given the data (assuming a uniform prior probability distribution over the DAG space) [115]. They relax the faithfulness assumption by replacing conditional independence tests with the goodness-of-fit tests. They are computationally expensive, as they enumerate (and score) every possible graph for the given variables. Well known score-based algorithms are GES [22] and its successor FGES [150], that use parallelization to optimize performance, and GFCI, that combines FCI and FGES [133].

FCM-based algorithms determine the causal direction of edges, identifying the true causal structure out of all the graphs within a Markov equivalence class. The most noticeable example, working for continuous variables, is LiNGAM, proposed by Shimizu *et al.* [163], where the model is assumed to be linear and non-Gaussian. Under the causal Markov assumption, acyclicity and a linear non-Gaussian parameterization (i.e., each variable is determined by a linear function of the values of its parents and an additive non-Gaussian noise term), it has been proved that the causal

structure can be uniquely determined [167]. FCM-based algorithms have the advantage of not relying on the faithfulness assumption and of learning substantially more about the causal structure, sometimes even determining a unique model. However, to relax faithfulness, they introduce other assumptions that can be controversial or difficult to test [115] and generally require a larger sample size to be accurate.

Continuous optimization-based algorithms combine the benefits of score-based methods with gradient optimization by converting the discrete DAG search space into a continuous and optimizable constraint space [187]. Usually, these methods are machine learning-based and the most representative ones are NOTEARS [205] and DAG-GNN [198].

Many CD algorithms allow to specify some constraints (e.g., required or forbidden causal relationships), based on prior human knowledge. The category of Hybrid or Prior Knowledge-based methods like Joint Causal Inference (JCI) [124] and Max-Min Hill-Climbing (MMHC) [182] aims to build on top of these algorithms and incorporate specific domain knowledge in addition to observational data in an effective way.

CD for time series. The conditional-based algorithms like PC, FCI, LiNGAM, although applicable to extract causal relationships between time series, are not specifically designed for this aim. Many extensions and specialized algorithms have been developed to address this specific task [125]. Among these, a strategy relying on Granger causality [51] and its generalization based on (Multivariate) Transfer Entropy [9] are common choices. Given two time series X and Y, the idea behind Granger causality is to investigate whether the prediction of the current value of time series X improves by incorporating Y's past into its own past [125]. In this case, we say that Y *Granger causes* X.

Extensive discussions on CD algorithms have been published by Glymour *et al.* [49], Nogueira *et al.* [132], Guo *et al.* [53], Wang *et al.* [187], and Moraffah *et al.* [125].

3.2 Causal Reasoning for Software Testing

CR is gaining increasing interest as an effective methodology to solve many and diverse SQA tasks, like Fault and KPI prediction [54, 105], Anomaly detection and Fault localization [199, 24, 123], and Threats mod-

eling [159]. Besides these, the ability of CR to determine the causal relationships between variables makes it particularly suited to run testing and analysis activities during the V&V phase of software systems life cycle. Exploiting causal relations can help execute tests more prone to spot defects of the system under test, with reference to both traditional software systems and learning-enabled ones. This aspect has been also underlined by Netflix, which included CR in their science-centric experimentation platform [34]. For instance, Pietrantuono *et al.* [145] use transfer entropy to uncover causal relationships between time series with the purpose of testing security of Internet of Things systems. Ji *et al.* [80] introduce a causality-aware coverage criterion for Deep Neural Networks with the aid of CD and with the SCM framework; coverage is computed by comparing the graph generated by different test suites to a ground truth graph. Ji *et al.*[79] use causality analysis to explore trade-offs between fairness and other key metrics in machine learning pipelines. They apply CD to reveal how fairness-improving methods affect performance and robustness and Double ML to compute the ATE. Rahman *et al.* [149] applies causality in deep learning models for vulnerability detection by using do-calculus to eliminate spurious features, like variable names and API names, which can lead to incorrect predictions. CR has been used also in the context of Autonomous Driving Systems: Maier *et al.* [113] employ SCM to predict the outcome of the simulation of driving scenarios (e.g., based on driving and/or environmental conditions) to expose critical scenarios; Zhong *et al.* [206] use CausalDAG to test multi-sensor fusion systems and verify that detected failures are caused by incorrect sensor fusion logic.

Regarding test generation, CR has been adopted by Oh *et al.* [134] to generate mutants in mutation testing, and by Clark *et al.* [24] to generate metamorphic relations in metamorphic testing. Additionally, CR is also adopted to evaluate and to improve the testing quality. Specifically, Li *et al.* [100] applies the natural experiment method to empirically investigate the factors affecting testing quality. Clark *et al.* [23] presents a testing framework incorporating a causal model into the testing process, enabling the direct application of CI techniques to software testing problems. Liu *et al.* [103] propose a technique concerning testing and analysis using causal graphs for Pointer Analysis. Finally, while most of the papers at the intersection between software testing and CR are employed in the V&V phase

of the software lifecycle, Fischbach *et al.* [44] concerns testing outside the V&V phase. The authors extract causal relations from the requirements to enable the automatic derivation of test cases and requirements dependencies.

While CR is gaining increasing interest in the software testing community, further research is required to enhance the maturity and readiness of CR-based solutions for industrial applications. Key areas that require attention include the development of scalable, robust tools and frameworks that can be easily integrated into existing testing workflows.

3.3 RBST Strategy

Established that the tester's question *"What happens to the test output Y if we fix the test input $X = x$?"* is inherently *causal*, a learning-from-association paradigm like ML cannot correctly answer it, as they do not handle causality [140]. While ML largely amplifies our pattern search ability, which is a great added-value, we can do much more, and rather amplify our causal reasoning ability. The latter is not limited to learning from observations, but it learns from hypothesizing *interventions* on variables of interest. Reformulating testing objectives as causal questions is possible given the underlying causal structure of the SUT. Consider the example context of test data generation, wherein the aim is to identify input combinations that maximize/minimize the test output (e.g., performance metrics or safety-related variables) or a metric of interest (e.g, coverage). In this context, cause-effect relations among variables described by a causal framework, explicitly give the possibility to query the model with the same tester's *causal* question and provide an accurate answer to it. This allows effective exploration of the test space by using the model, without executing any tests, so as to select and actually run only the most promising tests. Besides this example, CR can assist testers in many other tasks such as regression test selection and prioritization, test suite minimization, and test planning, or even to analyze results of test execution (and support debugging) with the use of counterfactuals (i.e.: as *Would have we still observed the failure if we had fixed $X = x$?*) This thesis will mainly consider the task of **test data generation**.

Efficient test case generation requires the ability to search the input

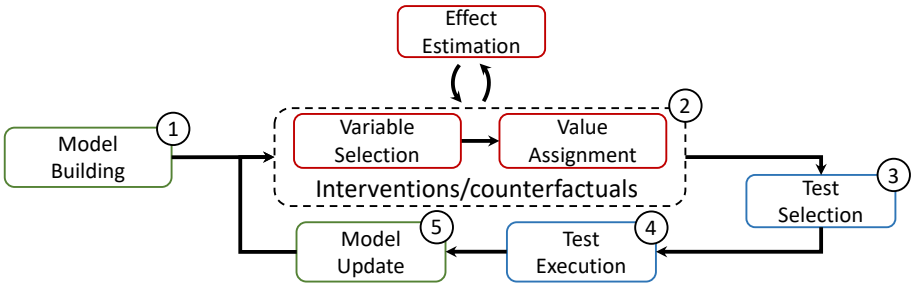


Figure 3.4. Reasoning-Based Software Testing

space intelligently, so as to identify combinations satisfying the testing goal (e.g., maximize fault detection or coverage) with reasonable cost. This exploration process can be redefined as a causal reasoning task. The envisaged Reasoning-Based Software Testing process is depicted in Figure 3.4).

The process starts from building a causal model ①, which we will consider as a graphical causal model (e.g., SCM) although any is suitable. It encodes the causal structure knowledge enabling the inference and is built initially and iteratively refined as more data becomes available. As mentioned in Section 3.1.3, controlled (or *soft*) experiments, CD from data (such as past executions of an initial bunch of tests), and the use of domain knowledge are the available options. It is worth stressing that, as in the latter case, integrating CD with additional knowledge sources (e.g., human expertise or architectural frameworks like service-dependency graphs in service-based systems) could significantly enhance the extraction of causal models that better align with the observed data. Additionally, this integration would allow for the validation of assumptions in certain application contexts where there is a lack of knowledge about the data-generating process (e.g., while two variables may be assumed causally independent, a causal discovery algorithm might reveal dependencies based on the data) [132].

Once the model is defined, it can be queried via a set of *interventions* and/or *counterfactuals* ②. They allow estimating the effect of a potential change and produce an answer to the query. The answer is actually a future (i.e., intervention) or past (i.e., counterfactual) *hypothetical test*, in which the value of one or more variables is actively set to a different

value from those observed. In test generation, this means generating a hypothesis for a test along with the expected effect if such test would be executed (or would have been executed, under different conditions). The RBST user here is required to develop a strategy for selecting the variable(s) on which to intervene, and the values to assign. More formally, let $V = X \cup Y = \{x_1, \ldots, x_m; y_1, \ldots y_n\}$ be the set of $m + n$ variables to define a test scenario, where X is the set of input variables and Y is the set of output variables. A strategy consists of (iteratively) selecting a variable $x_i \in X$, a value for it, and then observing the hypothetical output $\hat{y}_j \in Y$. In doing so, several strategies can be applied, depending on the testing purpose. For instance, an option is to select the intervention maximizing the information gained (i.e., minimize the uncertainty) about the true graph – hence intervening to better learn the graph [169]. On the other hand, testers might want to directly set an intervention maximizing/minimizing the desired test objective(s) (e.g., increase coverage, produce critical outputs) or maximize diversity (e.g., impacting more effects together, helpful in multi/many-objective testing). A combination thereof can be set up (also depending on the testing budget), e.g., to trade model accuracy and reward: initially, uncertainty-driven interventions improving the model could be better, gradually replaced by objective-driven intervention.

Regardless of the chosen criterion, both the variable and value selection/assignment can be implemented in several ways, such as: via probabilistic sampling (e.g., (non-)uniform random sampling), search-based or learning-based techniques, adaptive strategies (i.e., using previous selections to drive the next ones), or even exhaustively (depending on the context, the intervention computation time could be negligible compared to real tests execution time). Each intervention produces an effect, which can be seen as the outcome of the execution of the hypothetical scenario on the output variables of interest. The effect is typically quantified by the ATE, although other metrics can be of interest (e.g.: ATT, CATE) (Section 3.1.2 [132]), by using libraries such as `DoWhy` [161], analytically or via simulation, namely by sampling from the post-intervention distributions and computing the desired effect estimate. In any case, by querying the model via interventions/counterfactuals, a set of hypothetical test scenarios $H = \{\boldsymbol{h_1}, \boldsymbol{h_2}, \ldots, \boldsymbol{h_q}, \ldots\}$ is generated, where q indexes the query, and

the output of the query $\boldsymbol{h_q}$ is an estimate of the expected value for each output variable $\hat{\boldsymbol{y}}_q$, quantified through causal metrics.

This process of generating hypothetical tests can be extended to stateful testing, where a test case consists of a sequence of inputs across multiple steps. In stateful scenarios, each test case is defined as a sequence of steps $\boldsymbol{t_1}, \boldsymbol{t_2}, \ldots, \boldsymbol{t_k}, \ldots$, where each step $\boldsymbol{t_k} = (\boldsymbol{x_k}; \boldsymbol{y_k}) = (x_{k_1}, \ldots, x_{k_m}; y_{k_1}, \ldots, y_{k_n})$ represents the k-th input-output pair in the sequence and includes the values of variables $\boldsymbol{x_k}$ and corresponding outputs $\boldsymbol{y_k}$ at that step. Here, the user selects values for inputs in X at each step and may generate *hypothethical* steps or even entire tests as sequences of interventions that progressively update the test case state. The model's response to each intervention can then be observed through changes in the output variables in Y, allowing the user to analyze cumulative effects over the entire sequence.

From the so-obtained set of *hypothetical test suite*, one (or more) actual scenarios are selected ③ to be used. The basic choice is to get maximum-effect test(s), but alternatives are worth to be explored, e.g., to improve diversity or exhaustively select all the tests.

Once the scenarios are selected, they can actually be executed on the SUT ④. In test generation, this would mean giving the SUT the selected inputs and gaining the *actual* outcome, not the hypothetical given by querying the causal model. Here, the difference between the predicted outcome and the *observed* one (i.e., after the execution) gives the user a measure of the accuracy of the causal model. A number of traditional metrics can be used to quantify this estimate (e.g., distance metrics). In any case, the information gained with the actual execution serves to enrich the knowledge to update and refine the model ⑤. Finally, the updated knowledge can be used to double-check if the estimated effect is significant or not (i.e., if are due to chance), via refutation tests and confidence interval computation [161].

There is a wide range of possibilities to instantiate the process. Possible alternatives for the key steps of RBST are listed and summarized in Table 3.1, with a non-exhaustive list of alternatives. The selection of a combination of alternatives forms a test generation strategy.

Table 3.1. Alternatives in instantiating the RBST process

Step	**Description**	**Alternatives**
Model building	How to build the causal model	Causal Discovery Domain Knowledge Controlled Experiments
Intervention variable selection	How to select the variable(s) for the intervention	*Criteria:* Uncertainty, Confidence Test objective(s) Diversity *Strategies:* Sampling Search-based Adaptive Learning-based Exhaustive
Intervention value assignment	How to set the intervention value for the selected variable(s)	
Effect estimation	How to estimate the effects of an intervention	Analytic Simulation-based

Chapter 4

Instantiating RBST for stateless testing

4.1 Autonomous driving systems testing

Autonomous Driving Systems (ADS) for self-driving cars are learning-enabled systems based on Deep Neural Networks (DNNs), able to sense the environment and make decisions to drive the car safely with little or no human driver input [170]. Their spread is expected to increase in the upcoming years: a recent report projects the self-driving cars market size to surpass USD 65 million by 2030, expanding growth at a rate of 13.38% over 2022 to 2030 [1]. For people to justifiably trust ADS (and for companies to sustainably develop them), a fundamental challenge is how to effectively and efficiently test their behavior.

A primary focus of researchers is on testing decision-making DNNs as standalone components [152], based on datasets obtained without involving the DNNs under test. This is referred to as *model-level* testing [170] or *offline* testing [58]. Several such strategies have been proposed, including: DeepXplore [142], DeepRoad [204], DeepTest [178], DeepGauge [110]. Though important, model-level testing inherently misses the ability to spot system-level misbehaviors; e.g., negligible DNN mispredictions can accrue over time and expose failures despite high DNN accuracy [170], [58], [57].

For more realistic tests, *system-level* testing, also called *online testing*, is applied rather than (or beside) *model-level testing* [46, 114, 183, 6, 55]. In

this case, the DNNs are embedded into a simulated driving environment and tested in a closed-loop mode in interaction with the environment. System-level testing is able to account for the effect that the DNN predictions have on the environment and on the behavior of the whole system. However, despite recent efforts [170, 13, 12, 58, 55, 46, 153, 114, 183], it remains challenging to generate safety-critical test scenarios in an efficient way [55]. The space of all possible tests for the whole system, considering the input from all sensors such as cameras, LiDAR, and GPS, is very large. In addition, the difficulty in exploring this space is exacerbated by the need to test multiple safety requirements at the same time, such as distance from objects, from pedestrians, and from the center of the lane, which entails a multi- or many-objective search. Moreover, the evaluation of every generated test scenario is expensive: the use of high-fidelity simulators indeed reduces the cost of a real-world large-scale testing process (and is of course safer), but running a single test scenario takes several minutes and is computationally expensive.

The combination of these challenges requires a testing strategy that intelligently explores the space of all possible tests and generates only the most promising ones to expose critical behaviors.

4.2 Stateless testing of autonomous driving systems

Model-level (or *offline*) testing of ADS has been used extensively for testing the individual DNNs by either using adversarial examples (e.g., corrupted images) [142, 110, 204, 111, 207] and by using Generative Adversarial Networks (GANs) to perturb the input [204, 87, 137, 196, 192]. This type of testing is supported by coverage criteria such as neuron coverage [142, 110], combinatorial coverage [112], "discrepancy" between training/-validation data and test data [83, 194]. Besides exposing mispredictions, a different goal is to sample a minimal subset from the operational inputs set that, once manually labeled, can closely assess the accuracy [102, 52, 20].

Model-level testing does not test the whole system in its environment. Recently, *system-level* (or *online*) testing has been more widely investigated. On this line, researchers proposed solutions to generate scenarios that cause the system to misbehave [6, 58, 170, 46, 153, 114, 183, 55]; sce-

narios are generated in a stateless setting, meaning that the configuration on the simulated environment is fully pre-selected and then the ADS is left to drive through a certain route. In other words, the environment is not dynamically manipulated during the execution of the scenario, making test inputs independent of each other.

Gambi *et al.* present `AsFAULT` [46], a genetic algorithm combined with procedural content generation – a technique employed in video games for the automatic creation of virtual environments [179] - to generate virtual roads causing the ego vehicle to depart from the center of the lane.

`DeepJanus` [153] is a search-based tool that generates frontier inputs, i.e., similar input pairs that cause the ADS to mispredict for one input and work fine for the other one.

Tuncali *et al.* present `SIM-ATAV` [183] that combines combinatorial testing with requirements falsification; they use covering array as a combinatorial test generation approach for discrete variables, and a falsification approach using uniform random search or, again, a search-based algorithm (simulated annealing) to search over continuous variables.

Majumdar *et al.* propose `Paracosm` [114], a simulation-based testing language, associated with a tool, that generates tests using random sampling for discrete parameters, and deterministic quasi-Monte Carlo methods [130] for continuous parameters to achieve high diversity.

Klishat and Althoff [85] propose an approach for testing of motion planning algorithms in ADS; it automatically generates critical scenarios based on a minimization of the solution space of the vehicle under test via evolutionary algorithms. Calò *et al.* [19] also use search-based techniques aiming at finding *avoidable* collision scenarios (i.e., scenarios in which the collision would not have occurred with a reconfiguration of the ADS). They first search for a collision and then for an alternative configuration of the ADS which avoids it.

Li *et al.* [99] present `AV-FUZZER` a framework aiming at finding single-objective safety violations by perturbing driving maneuvers of traffic participants (i.e., acceleration/deceleration, following lane, and making lane change). Metamorphic testing is also used in combination with equivalent partition testing for system-level ADS testing [138].

Other studies focus on system-level testing, but not with the objective of generating test cases. Stocco *et al.* [170] compare virtual and physical-

world system-level testing; in [172], the same authors propose an oracle for mispredictions detection; in [171], the authors use knowledge inferred from field execution to predict misbehaviors; Haq *et al.* compare online and offline testing [58].

The above studies pursue testing efficiency by trying to tackle the large-state-space challenge. None of them focuses on the additional challenges brought by the need to consider many safety requirements at the same time, which entails a many-objective search. Abdessalem *et al.* [6] use ML models (decision trees) combined with the NSGA-II search-based multi-objective algorithm to guide the search – three objectives are considered. The same authors [13] are the first ones to formulate the ADS testing problem as a many-objective search. Luo *et al.* propose EMOOD, an approach that uses an evolutionary algorithm to generate test scenarios to expose as many combinations of requirements violations as possible [109]. Haq *et al.* [55] adopt the same formulation, and address the expensive test case evaluation challenge too. They propose SAMOTA, a technique that uses surrogate models (based on regression and radial basis function networks) to predict the outcome of a test case without actually executing it. Similarly to SAMOTA but out of the ADS testing domain, the use of surrogate models to address computationally expensive optimization problems has been widely studied [81], with recent studies combining global and local search [208] and also using the most uncertain candidates in addition to the best predicted ones[186, 104].

A few of these works have the objective of exploiting some learned relationships between test input and output, and using them to generate only interesting (i.e., safety-violating) tests. However, none of them employ causality to achieve this objective. An instance of RBST, called CART, is presented in the following. The key advancement of CART is to inject CR into the test generation process. For what was discussed, this is a more versatile tool, since causation is more informative than correlation, and learning cause-effect relations enables a tester to spot more precisely input values more likely to cause a safety violation.

4.3 Problem definition and notation

The goal of online testing of ADS is to generate a minimal set of driving scenarios to be run on a simulator – called *test scenario* hereafter – that causes the system to violate multiple safety requirements. This is a many-objective optimization problem.

Usually, the ADS is embedded within a simulator, like CARLA [36], which renders a realistic town environment including junctions, vehicles, pedestrians, traffic lights, and traffic signs. The ADS that controls the **ego-vehicle** (**EV**) has to drive it through a predefined route. At each time step, the ADS receives and processes data from sensors (e.g., camera and LIDAR) to generate driving commands (steering, throttle, and braking) to maximize the driving performance. The CARLA leaderboard [93] measures the ADS performance with the *driving score*, a combination of two metrics, namely route completion and infraction penalty. The former is the percentage of route completion; the latter measures the number of infractions, including traffic rules violations (e.g., red lights and stop signs) and detected collisions with other vehicles, pedestrians, and static elements (e.g., road signs).

In this context, it is possible to define six functional and safety requirements [55] for the ADS (listed in Table 4.1): the EV must *(r1)* keep the lane; it must not collide *(r2)* with the VIF, *(r3)* with the pedestrian, and *(r4)* with static meshes (e.g., traffic lights/signs); it must *(r5)* complete the route within the given time, and *(r6)* abide by traffic rules.

The simulator reports violations of the requirements, respectively: if the **distance from the center of the lane** (**DCL**) exceeds a threshold identifying the lane boundaries; if the **distance from the VIF** (**DV**), or the **distance from the pedestrian (DP)**, or **distance from static**

Table 4.1. Safety and functional requirements for the ADS under test.

Req.	Description	Metric
$r1$	EV must keep the lane	EV distance from center of lane
$r2$	EV must not collide with other vehicles	EV distance from other vehicles
$r3$	EV must not collide with pedestrians	EV distance from other pedestrians
$r4$	EV must not collide with static meshes	EV distance from static meshes
$r5$	EV must complete the route	EV distance from destination
$r6$	EV must abide by traffic rules (i.e., red lights)	Flag detecting red light violation

meshes (DS) is less than or equal to zero (i.e., a collision occurs); if at the end of the scenario the **distance from the destination (DT)** is greater than zero, and if it detects that the EV has violated a **traffic rule (TR)** (e.g., running a red light[1]). In the simplified example introduced in previous chapters (i.e., Figure 3.1) with 2 input variables (i.e., "Vehicle target speed" and "Road type") and a single requirement (i.e., "Distance from center of the lane"), the goal is to find the combinations of the two inputs that cause a violation of the requirement.

More formally, consider the following notation:

- $V = X \cup Y = \{x_1, \ldots, x_m; y_1, \ldots y_n\}$ is the set of $m + n$ variables to define a test scenario, where X is the set of input variables and Y is the set of output variables.

- $\boldsymbol{s}_k = (\boldsymbol{x}_k;\ \boldsymbol{y}_k) = (x_{k_1}, \ldots x_{k_m}; y_{k_1}, \ldots y_{k_n})$ is the k-th *test scenario* (or *test*) where $\boldsymbol{x}_k$ (input) and $\boldsymbol{y}_k$ (output) are the values of the corresponding variables in X and Y taken in the k-th test.[2]

- D is a *database* containing all the executed tests.

- $Thr = (\tau_1, \ldots, \tau_n)$ is the set of thresholds associated to output variables in Y, which allows determining a safety violation occurred or not.

- $R = \{r_1, \ldots, r_n\}$ is the set of safety requirements associated with the n output variables and the corresponding thresholds.[3] A safety requirement r_j is violated when $y_j < \tau_j$ at any time during the simulation. Without loss of generality, assume that the lower the values of output variables, the closer the requirement violation (e.g., if a "distance from vehicles" output gets close to a threshold of 0.5

[1] Running a red light is the often the only violation considered for TR [55, 56].

[2] This is similar to, but not to be confused with, the stateful formulation of RBST (Section 3.3). Here, there is not a dynamic sequence of states, each test case is assumed to start from the same initial SUT state.

[3] For simplicity, assume that a safety requirement is associated with one measured output variable; clearly, the notation can be extended to deal with safety requirements defined on combinations of output variables. In this case $|R| < |Y|$, and a derived output variable expressing the desired combination is considered in the output variables set (e.g.: $y'_j = f(y_1, \ldots, y_n)$, with the associated threshold τ'_j).

meters, a safety requirement is close to being violated). When dealing with variables that are more critical when their value increases (such as "distance from the center of the lane"), its opposite is considered.

- $C \subseteq R$ ($U \subseteq R$) is the subset of safety requirements that have been *covered* (respectively: *uncovered*). Covering a requirement means that at least one test violates it.

Testing can be targeted at covering as many requirements as possible (i.e., maximize the proportion of covered safety requirements, $|C|/|R|$), or at exposing as many safety violations as possible, regardless of requirements already covered. The latter is because testers might want to have multiple diverse tests violating the same requirements that highlight different conditions in which the violation occurs.

4.4 The CART algorithm

Given the set of requirements to violate R, the CART algorithm (CAusal-Reasoning-driven Testing) addresses the many-objective optimization problem by a classical weighted-sum approach [31, 32], namely combining the many objectives into a single one, hereafter called *fitness* and expressed by a fitness function Φ. From the two above-mentioned testing goals, it is possible to consider two fitness functions. The former expresses the proportion of covered safety requirements and is defined as:

$$\Phi_A(\boldsymbol{s}_k) = \frac{\sum_{j=1}^{|U|}(1 - mM(\boldsymbol{y}_{k_j}))}{|U|} \tag{4.1}$$

where $mM(y_{k_j})$ is the min-Max normalized j-th output variable of the k-th test – lower values are closer to a safety violation - and U is the set of still uncovered safety requirements. This gives priority to tests more likely to affect the output variables related to still-uncovered requirements. Since the set of uncovered requirements changes during testing, this function is called *adaptive*.

The second fitness function maximizes the number of safety violations.

This is called *fixed*, and is defined as:

$$\Phi_F(\boldsymbol{s}_k) = \frac{\sum_{j=1}^{|R|}(1 - mM(\boldsymbol{y}_{k_j}))}{|R|} \tag{4.2}$$

where R is the set of safety requirements. In this case, the function gives priority to tests more likely to cause a safety violation of any safety requirement, even though some past tests already covered it.

Whenever possible, ϕ_k will refer to the value of any of the above fitness functions, for test scenario $\boldsymbol{s}_k$.

At a high level, the CART algorithm acts in two phases. In the first phase, CART extracts (and iteratively updates) a SCM from past data, via CD. Such data needs to be in the same format used to define a test case - one entry of the dataset consists of values of the input variables (which correspond to a specific test scenario) and of the obtained output variables in that scenario. In a testing process, this data would naturally come from previous testing sessions; the very first time that CART is executed, the initial set of tests can be obtained by random testing or any other technique. These data could also come from driving data; in such a case, the input and output variables for each scenario need to be extracted from such data (depending on how historical data are gathered and stored) and formatted as test scenarios. The resulting SCM describes the causal relations between the input and output variables of a test scenario (cf. with Definition 4). In the second phase, this model is queried by a CI engine, which runs interventional queries such as: *what is the effect on the output variable of interest if we set the input to a given value?* More formally, the CI engine assesses the expected value of the output variable of interest $y_j \in Y$ under a *do*-intervention on the input variable $x_i \in X$: $E[y_j|do(x_i = x)]$, where x is the hypothesized input value (cf. with Section 3.1.2). For instance, with reference to the example in Figure 3.1, it means generating queries such as: *what is the effect on "Distance from center of the lane if we set "Vehicle target speed" to 40km/h?* These queries produce a set of "*hypothetical*" tests whose output is an estimate of the expected effect on the output variables of interest under hypothetical inputs. No real test is executed in this phase. Actual real tests to run are then generated by selecting only the best tests from the hypothetical test set. The core idea is to explore the test search space by causal queries to the model

Algorithm 1 CART algorithm

Input: D: database of tests; f: boolean to select the adaptive (0) or fixed (1) fitness function; R: safety requirements; Thr: error thresholds; η_0: CI sample size; ϵ: probability for input selection.
Output: S: Test suite, M: causal model, D: Updated database

$S, P \leftarrow \emptyset$
repeat
  $M \leftarrow buildCausalModel(D)$
  $P \leftarrow updatePopulation(D, T)$
  $T \leftarrow \emptyset$
  for $k = 1$ to $|P|$ **do**
    $\tilde{H} \leftarrow infer(M, \eta_0, P_k, \epsilon)$ Query to the model. $\tilde{H}$: set of hypothetical tests
    $H \leftarrow estimateFitness(\tilde{H}, f, R, Thr)$
    $\tilde{t} \leftarrow select(H)$ t: best among $|H|$ hypothetical tests
    $t \leftarrow runOnSimulator(\tilde{t}, R, Thr)$ Compute actual fitness
    $T \leftarrow T \cup t$
  end for
  $(S, D) \leftarrow (S, D) \cup T$
until !$terminatingCondition$
return S, M, D

rather than by actually executing the tests (which in contexts like ADS takes several minutes per test).

Algorithm 1 describes the CART steps for test generation. The algorithm takes, as input, the set of safety requirements R to cover with the associated thresholds E, the database D containing a set of already executed scenarios (e.g., derived from past tests or from driving data), the sample size parameter η_0 used by the Causal Inference (CI) engine as explained below, a flag to select the fitness function, and a parameter ϵ, a probability value for inputs selection policy. It returns the generated test suite S, the updated database D, and the final causal model M, refined over successive iterations. The termination condition could be the exhaustion of the budgeted testing time, or a convergence criterion (e.g.,

violations no longer found after some iterations). The detailed CART steps follow:

1. **Causal model building**. A SCM is inferred (line 3) from data contained in D. The SCM is derived by first inferring the graph structure via a CD algorithm (cf. with Section 3.1.3). In the proposed implementation, five CD algorithms are tested from the `Pycausal` library based on `Tetrad`. Then, the stochastic models and FCMs for, respectively, root and non-root nodes are assigned to each variable based on data. Specifically, by using the CI library `DoWhy`, CART assigns the best-fitting stochastic model among linear, polynomial, and gradient boost, and the best-fitting FCMs among linear and non-linear additive noise models [162]. `DoWhy` supports a variety of other models, also from the `Sklearn` and `Scipy` libraries; the default configuration is adopted.

   The so-derived model is refined at every iteration with the updated database D including the scenarios executed in the previous iteration (line 13).

   It is worth stressing that domain knowledge can be leveraged to build or refine the graph – one of the advantages of causal models. This would however require human intervention and domain expertise; for the sake of full automation, this possibility is ignored.

2. **CI and test generation**. Lines 6-12 generate tests starting from the population of current tests P and the (updated) causal model M. Tests are derived according to an evolutionary strategy, namely by trying to improve the current population of tests. The population contains at every iteration the top-$|P|$ tests from $D \cup T$, where T is the set of tests selected and executed in the previous iteration (initially empty). The population size is fixed, with $|P|=|R|$ as in previous studies [13], [55]. For every test $\boldsymbol{s}_k$ in P, denoted as P_k, the causal model M is queried in order to generate multiple hypothetical tests, estimate their expected fitness (line 7-8), and take the best one (line 9-10). The following steps are carried out:

   - *Input variable selection.* A query is an intervention on an input variable $x_i \in X$ that changes its value to assess the ef-

fect on output variables. Therefore, an input variable needs to be selected first. This means, with reference to the simplified example shown in Figure 3.1, to select one of the two input variables (i.e. "Vehicle target speed" and "Road type"). CART selects with probability ϵ, the input variable in the SCM with the greatest out-degree (counting only edges toward the output variables $y_j \in Y$), with ties broken randomly, since it is the variable expected to impact more safety requirements together (in the example the two input variables have the same out-degree, resulting in a random choice); with probability $1 - \epsilon$, the input variable is selected randomly (all the variables having the same probability) so as to promote diversity. Thus, a low value of ϵ gives higher diversity, since the input variable on which to intervene would be chosen randomly, scarcely exploiting the knowledge encoded into the model, in favor of exploration. Contrarily, a high value of ϵ leads to choosing, with higher probability, the input variable expected to causally impact more safety requirements together. In the evaluation, $\epsilon = 0.5$ to balance exploration and exploitation and avoid bias toward one of them.

- *Intervention value assignment.* For each P_k, a set of hypothetical test scenarios $\tilde{H} = \{\tilde{\boldsymbol{h}}_1, \tilde{\boldsymbol{h}}_2, \dots, \tilde{\boldsymbol{h}}_q, \dots\}$ is generated by querying the model via interventions on the selected x_i (with q indexing the intervention). The intervention (i.e., setting a value for the selected variable) causes a change in the other variables' distributions directly or indirectly related to it. CART uses the simulation-based inference (cf. with Section 3.1.2) to estimate the expected effect of the intervention on the output variables, using the `DoWhy-GCM` library [16]. Simulation-based inference draws samples from the post-intervention distributions, namely it uses the SCM after the intervention with the associated distributions, and draws samples from it. Specifically, it *i)* sorts the nodes in topological order, *ii)* sample values from root nodes according to their distribution, and then *iii)* use the structural equations with randomly sampled noise to compute the values for downstream nodes. The values for those variables not present in the SCM (i.e., with no cause-effect re-

lation with any other variable) are kept with the same value of the P_k test, given as input to the method *infer*. With reference to Figure 3.1b, the variable "Vehicle Target Speed" is selected for the intervention; thus the inferential engine fixes the value of this variable, samples from the distribution of "Road type" (since it is a root node), and propagate the sampled values to the only downstream node, which is "Distance from Center of the Lane", computing new data with the structural equation. The engine draws, for every intervention, samples of size $\eta = \eta_0$ ($\eta_0 = 1,000$ is the default value of `DoWhy-GCM`), from which the expected values of the output variables are taken as post-intervention estimates (denoted as $\hat{\boldsymbol{y}}_q$ for the q-th intervention). It is worth stressing that these hypothetical tests are queries done to the causal model, and are not scenarios actually executed on the simulator. The output of the query, $\hat{\boldsymbol{y}}_q$, are estimate of the expected value for each output variable.

As for the value to assign to the intervention variable, several policies are possible (e.g., uniform or non-uniform random sampling, adaptive strategies, learning-based or search-based criteria), also depending on the type of variables, which could be continuous, discrete, or mixed. Since, in this case study, there are only discrete input variables, a query (i.e., do an intervention) is run for each value of the selected variable. The final number of hypothetical tests ($\tilde{H}$) equals the number of interventions. This number can be reduced if needed (for instance, in the presence of continuous variables) by the mentioned policies. In the evaluation, its impact turned out to be negligible compared to test execution time (see Section 4.6.5).

- *Fitness estimation and scenario selection.* The fitness for each hypothetical test $\tilde{\boldsymbol{h}}_q \in \tilde{H}$ is estimated by applying the fitness function (which requires the set of requirements and thresholds, see Eq. 4.1 and 4.2) to the generated samples, hence to the output variable estimates, $\hat{\boldsymbol{y}}_q$, as computed by the CI query. Thus, the `estimateFitness()` at line 8 uses $\hat{\boldsymbol{y}}_q$ to compute an estimate of the fitness, using one of the two fitness functions (fixed or adaptive, Eq. 4.1 and 4.2, selected via the boolean f).

Table 4.2. RBST instantiation for CART

Step	**Choices**
Model building	CD
Intervention variable selection	Node out-degree Random
Intervention value assignment	Exhaustive
Effect estimation	Simulation-based

This gives the set $H = \{\boldsymbol{h_1}, \boldsymbol{h_2}, \ldots, \boldsymbol{h_q}, \ldots\}$, with the hypothetical tests $\tilde{\boldsymbol{h}}_q$ along with their fitness estimate $\hat{\phi}_q$ (namely: $\boldsymbol{h_q} = \tilde{\boldsymbol{h}}_q \cup \hat{\phi}_q$), line 8. Only the hypothetical test with the highest $\hat{\phi}_q$ is selected (line 9) and then executed on the simulator to compute its actual fitness ϕ_q (line 10), producing the actual test scenario t. This is added to the set of tests generated in that iteration, T.

The set of so-generated tests is added to both the test suite S and the database D. Note that both S and D are updated in the same way (i.e., by adding T), therefore S can be obtained by simply taking D at the end of the algorithm and removing tests in the initial D given as input. However, for the sake of clarity, both are kept in the pseudo-code. The output also includes the causal model M, which is a valuable tool for future tests or even other engineering tasks.

The CART formulation constitutes a RBST instance, summarized in Table 5.2. The base *criteria* is to find tests maximizing the testing objectives, while the *strategies* for model building, variable selection, value assignment, and effect estimation are, respectively, CD, Node out-degree, or random (depending on ϵ), exhaustive, and simulation.

4.5 Evaluation

4.5.1 Compared techniques

CART is compared to random search and to the following state-of-art techniques for many-objective search-based testing: MOSA [136], FITEST [13] and SAMOTA, all used to test ADS in the recent study presenting SAMOTA [55]. Like CART in the adaptive-fitness configuration, these techniques generate a population of solutions aiming to cover a higher proportion of safety requirements. CART, however, addresses the many-objective optimization problem by a classical weighted-sum approach [31, 32], with the normalized objectives having equal weights – cf. with Section 4.4. The combined single objective is used in the test generation process.

MOSA was first used to formulate branch coverage as a many-objective optimization problem. FITEST later extended it, by progressively updating the fitness function marking the already covered requirements as testing progresses, thus reducing the population size and improving efficiency. SAMOTA holds the features of FITEST, but exploits surrogate models to estimate the fitness function without actually executing tests. SAMOTA is the technique closest to CART, as its surrogate models are meant to avoid non-promising simulator runs. However, CART uses causal models to this aim in order to support the generation of effective test scenarios.

The mutation and crossover parameters of SAMOTA, FITEST and MOSA are set at the default values used in [55] and in [136] (they all have mutation rate $\frac{1}{psize}$; crossover rate is 0.75 for MOSA and SAMOTA, 0.60 for FITEST). Both SAMOTA configurations are used (SAMOTA-I with the initial database, and SAMOTA-E with the empty database). The implementation of SAMOTA, FITEST and MOSA are taken from the replication package[4] provided by Haq *et al.* [55]. The random algorithm (called RANDOM hereafter) samples inputs uniformly within their lower-upper bounds.

4.5.2 Research Questions

- **RQ1** (*Usefulness*). Does CR support the generation of test scenarios?

[4] `https://doi.org/10.6084/m9.figshare.16468530.`

  - *RQ1.1.* How does CI perform in generating representative test scenarios?
  - *RQ1.2.* How do different CD algorithms perform?

- **RQ2** *(Coverage of safety requirements).* How do techniques perform in covering safety requirements?
  - *RQ2.1.* How *effective* are the techniques in covering safety requirements?
  - *RQ2.2.* How *efficient* are the techniques in covering safety requirements?

- **RQ3** *(Detection of safety violations).* How do techniques perform in detecting safety violations?
  - *RQ3.1.* How *effective* are the techniques in exposing safety violations?
  - *RQ3.2.* How *efficient* are the techniques in exposing safety violations?

- **RQ4** *(Test suite quality)* What are the fitness and diversity of the generated test suites?

In CART, the CI engine generates what was called "hypothetical" test scenarios, i.e., a combination of test inputs that are not actually run on the simulator to get the output, but are instead used to query the causal model (i.e., to do an *intervention*) and get an estimate of the expected output. Therefore, before assessing CART against the baselines, RQ1 first aims to assess to what extent the expected output of such hypothetical test scenarios are close to the actual test output obtained by running the same test on the simulator. This ability would, by itself, pave the ground to new opportunities in testing, e.g., by exploiting the possibility of running *what happens if* queries to a model (i.e., what is the expected output if given a certain input) and predict tests outcome without actually running them. Furthermore, as different CD algorithms can provide different models, hence potentially different results, RQ1 compares 5 algorithms: PC, FGES, FCI, GFCI, RFCI.

RQ2 first investigates the proportion of safety requirements violated by at least one test scenario, given a fixed testing budget (RQ2.1); the goal is to have a minimal test suite that covers as many safety requirements as possible. Then, RQ2.2 evaluates the performance over testing time, so as to see which algorithm achieves the goal earlier.

RQ3 evaluates the ability of the compared techniques of generating *critical* (i.e., safety-violating) tests. The fitness function used in RQ2 is what we called the *adaptive* function Φ_A; this tracks the safety requirements that get covered during testing, so as to orient the test generation toward scenarios more likely to cover the remaining requirements.

Besides requirements coverage, testers might be interested in the number of safety-violating test cases (possibly with more different tests violating a requirement). In fact, having only one example for a violated safety requirement could be not satisfying for engineers; multiple diverse tests violating the same requirements highlight possible different conditions under which the violation occurs, each of which could trigger different faults leading to that violation. Thus, a richer set of safety-violating tests can support the root cause analysis and debugging task. This is pursued by what was called the *fixed* fitness function Φ_F, adopted in RQ3, which pushes toward safety-violating scenarios without looking at the already-covered safety requirements. To this aim, Φ_F have been implemented in CART and SAMOTA, FITEST and MOSA have been modified to make them ignore the set of still-uncovered requirements in their fitness function. These modified versions never update the set of covered requirements, always generating tests that try to cover all requirements.

RQ4 investigates the quality of the generated tests, considering their fitness and diversity. As for fitness, the interest here is toward in those test scenarios that, even though not causing a violation of the safety thresholds, push the variables of interest (e.g., *distance from pedestrians*) very close to a violation (i.e., "near-violating" test scenarios). Such tests highlight suspicious behaviours and potentially dangerous situations. For instance, a scenario in which the car stops very close to a pedestrian might be not desirable even though it is not a collision, and testers may wish to check them. Moreover, since all the compared techniques follow an evolutionary approach (which gradually improves solutions), it is also of interest to check which technique produces higher-fitness tests, as tests with high

fitness are more likely to evolve into safety-violating tests if more testing time is available. Finally, multiple similar tests can trigger the same violation or near-violation (especially under the fixed fitness function that can generates more tests for a requirement), while a typical desideratum of test suites is to have diverse tests – hence diversity of the test scenarios is also checked.

4.5.3 Experiment design

The case study used for the evaluation is Pylot [50], a well-known ADS with state-of-the-art techniques based on pre-trained DNNs. It has a natural compatibility with CARLA [36], and it scored as a top submission for CARLA Autonomous Driving Challenge. Pylot is also chosen because of its high adaptability to customized techniques. A test case is characterized by sixteen input variables, describing the road type, the presence of (possibly two-wheeled) vehicles in front, in adjacent or in opposite lane, weather conditions, presence of pedestrians, of trees, of buildings, speed (full list available in the replication package). Tests are generated with input within the boundaries of the simulator's domain, thus the scenarios are as required by the simulator.

As output, the six requirements in Table 4.1 are considered. Information about the output metrics is retrieved from the simulator. In particular, the distance from the center of the lane is computed as the distance between the center of the vehicle and waypoints that are typically placed in the center of the lane (when multiple lanes are available, the waypoints follow, in case of lane changes, the trajectory that crosses the lane, preventing a safety violation at each change); collisions are detected both by *collision events* and by collected distances (from vehicles, pedestrians, and static objects).

Three different experiments are run to answer RQ1, RQ2, and RQ3. The comparison done in RQ4 uses the test suites produced in RQ2 and RQ3 (with the adaptive and fixed fitness functions, respectively). For RQ1, 1,000 random test scenarios are executed to investigate causal models and causal inference performance. For RQ2, each compared technique is run 20 times to get statistically significant results (at significance value $\alpha = .05$). Every execution has a fixed time budget, which is 2 hours, in line with Haq *et al.* [55]. Moreover, for the techniques that use a database

of initial solutions (i.e., CART and SAMOTA-I), two databases of different size are considered: one with exactly the same 39 test scenarios used in the SAMOTA work [55], for a fair comparison, and one with 100 randomly generated test scenarios (called hereafter CART_{100} and SAMOTA-I_{100}). The 39 tests used by SAMOTA are obtained by a 4-way combinatorial coverage testing based on all the attributes used to define the test input space to generate diverse test cases. This results in eight techniques (CART_{100}, CART, SAMOTA-I_{100}, SAMOTA-I, SAMOTA-E, FITEST, MOSA, RANDOM). For RQ3, further 20 repetitions per technique are run, except RANDOM, with the changed fitness function (the *fixed* one), again with a time budget of 2 hours and with both databases of initial solutions. In fact, the RANDOM technique does not distinguish fixed form adaptive function; for it the same tests used in RQ2 are employed.

Overall, with 676 computing hours, the total number of tests is 4,457: 1,000 random generated tests for RQ1, plus 1,909 tests for RQ2 and 1,548 tests for RQ3 resulting from the 40 (20 + 20) runs for every techniques. All experiments were run on a virtual machine built on Google Cloud Compute Engine platform[5]. It was configured with Ubuntu 18.04 running on Intel Haswell CPU (4 cores) with NVIDIA Tesla T4 (16 GB) and 16 GB memory.[6] Further settings and RQ-dependent details are presented in the following Section.

4.6 Results

4.6.1 RQ1: Usefulness of causal models

RQ1 uses a set of $S = 1,000$ test scenarios generated by the RANDOM technique and executed on the simulator. The procedure is as follows: 5% of these tests are used to train the causal model (training size $z_{tr} =$ 50 randomly-sampled tests); then, from the remaining 950 test scenarios, further $z_{ts} = 50$ tests are randomly sampled as test set. 50 tests can be considered as a "relatively small" test set (i.e., implicitly assuming that, in a realistic environment, the number of tests easily gets to 50), so as to show that causal models can be useful even with few entries.

[5] `https://cloud.google.com/compute`.

[6] Google LLC, 2020. G Suite, Available at: `https://gsuite.google.com`.

Training and test sets are denoted as TrS and TS, respectively. The aim is to compare, for each of the 50 tests in the testing set, the real observed outputs (specifically, the fitness $\phi(\boldsymbol{s_k})$ of the test) against the prediction done by the causal inference engine querying the causal model: specifically, for each test $\boldsymbol{s}_k = (\boldsymbol{x}_k; \boldsymbol{y}_k) \in TS$, the CI engine queries the causal model to predict what is the output under the input $\boldsymbol{x}_k$, thus getting an estimate $\hat{\boldsymbol{y}}_k$. The fitness is computed on the real observed output, $\boldsymbol{y}_k$, and on the estimated one, $\hat{\boldsymbol{y}}_k$ (getting ϕ_k and $\hat{\phi}_k$, which are then compared to each other). In causal inference terms, this means predicting the effect of applying an intervention. To account for randomness, this process is repeated 20 times, with different training and test sets. The sample size for simulating the intervention is set to $\eta = 1,000$, which is the default value in the used library, `DoWhy`.

To compare the predicted *vs* the actual output, the following two metrics are used: the percentage Root Mean Squared Error (%RMSE) and the Rank Biased Overlap (RBO) for rankings comparison. The former is a well-known metric to compare different models' prediction, corresponding to the RMSE normalized by the RMS value of the predicted value:

$$\%RMSE = \sqrt{\frac{\frac{1}{n}\sum_{k=1}^{n}(\phi_k - \hat{\phi}_k)^2}{\sum_{k=1}^{n}\hat{\phi}_k^2}} \cdot 100 \tag{4.3}$$

where ϕ_k and $\hat{\phi}_k$ are the actual and predicted fitness value of the k-th test, respectively. This metric measures the accuracy of the fitness value prediction – the lower, the better.

A tester may be interested just in the ability of the model to distinguish critical tests, rather than in predicting the exact output value. Therefore the RBO is reported, a well-known metric to compute rankings similarity that, unlike Spearman's or Kendall's correlation, weighs the (dis)agreements on the top positions more than the ones at the bottom. The $t = 50$ tests are ranked by their actual fitness from more to less critical ones (list T), and by their predicted fitness (list P). The extrapolated RBO bounded at t is [190]:

$$RBO(A, P, q, t) = A_t \cdot q^t + \frac{1-q}{q}\sum_{d=1}^{t} A_d \cdot q^d \tag{4.4}$$

Table 4.3. RQ1 - CSD algorithms configuration

Algorithm	Parameters and values
FGES	scoreId = cg-bic-score, dataType = mixed, numCategoriesToDiscretize = 7, maxDegree = 3, faithfulnessAssumed = True, numberResampling = 5, resamplingEnsemble = 1, addOriginalDataset = True
PC	testId = fisher-z-test, fasRule = 2, depth = 2, conflictRule = 1, concurrentFAS = True, useMaxPOrientationHeuristic = True
GFCI	testId = cg-lr-test, scoreId = cg-bic-score, dataType = mixed, numCategoriesToDiscretize = 7, maxDegree = 3, maxPathLength = -1, completeRuleSetUsed = False, faithfulnessAssumed = True
FCI	testId = fisher-z-test, depth = -1, maxPathLength = -1, completeRuleSetUsed = False
RFCI	testId = cg-lr-test, dataType = mixed, numCategoriesToDiscretize = 7, depth = -1, maxPathLength = -1, discretize = False, completeRuleSetUsed = False, numberResampling = 5, resamplingEnsemble = 1, addOriginalDataset = True

where: $A_t = |T_{1:t} \cap P_{1:t}|/t$ is the proportion of the overlap of the $T_{1:t}$ and $P_{1:t}$ lists (with elements from position 1 to t) of the ranking to be examined; q is a parameter in (0, 1) that determines how steep the decline in weights is, given to the positions in the list – a smaller q gives more weight to the overlaps in top positions. $q = 0.98$ is used, giving the $t = 50$ ranks a weight of 86% [190]. In order to build the causal model with the $z_{tr} = 50$ randomly sample tests, five different CD algorithms are employed: PC, FGES, FCI, GFCI, RFCI. Their configuration (Table 4.3) is the default configuration in `Tetrad` [151] (and `pycausal`), used in various studies on CD [193], [205].

The violin plots in Figure 4.1 show the distribution of %RMSE and RBO. The %RMSE plot indicates that the prediction error done by querying the causal model is between 5% and 6%, regardless the adopted CSD algorithm. The good result is confirmed by the RBO metric, which is approximately between 0.6 and 0.7: this roughly means the two lists of tests (made by ranking the real observed vs predicted fitness) have 60%-70% of their ranks in common [190]. Also in this case, the CSD algorithm seems to not have an impact. To confirm this statistically, the Friedman test [45]

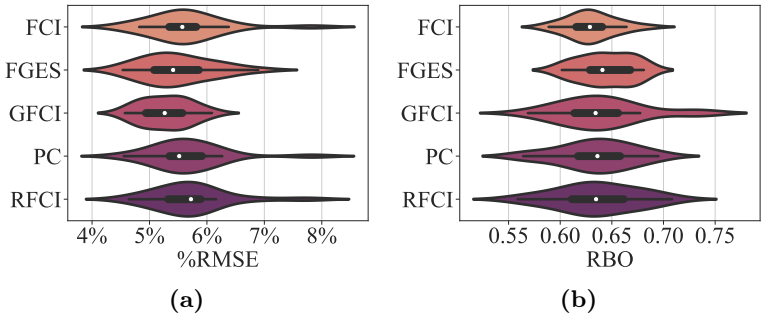


Figure 4.1. RQ1 - %RMSE and RBO of the 5 CSD algorithms

is run, a non-parametric hypothesis test for Analysis of Variance, which assesses if there is at least one technique that significantly differs from others.

For both %RMSE and RBO the null hypothesis of no significant difference between CD algorithms cannot be rejected (p-value equals 0.9655 for %RMSE and 0.2177 for RBO). Thus, the CD algorithms turned out to be statistically equivalent, any of them can be adopted in CART. In the next RQs, CART is configured with GFCI, since it has slightly smaller %RMSE and similar RBO to the others.

RQ1 answer

Causal inference is a useful tool for engineers to predict the effect of hypothetical inputs on target outputs; the observed error in the case study ranges from 5% to 6%.
The model predictions are also useful to rank the highest-fitness test first (e.g., for prioritizing tests), as the ranking similarity is between 0.6 and 0.7. This can enable the design of new testing techniques based on causal inference.
The CSD algorithm had no significant impact on the performance of the prediction, as both the prediction error and ranking on the give models built by the algorithms are statistically equivalent.

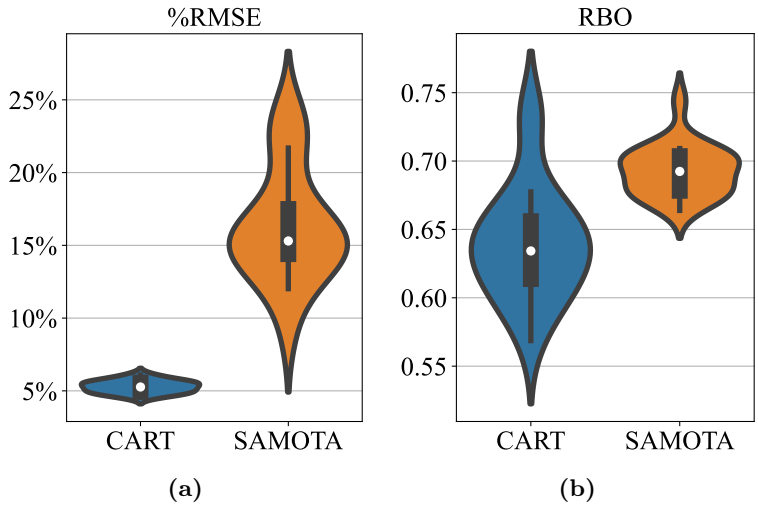


Figure 4.2. RQ1.1 - %RMSE and RBO: CART causal models and SAMOTA ML-based models

Comparing CART causal models against SAMOTA ML-based models

In the following, a comparison is made between the prediction made by causal models *vs* the prediction made by ML-based surrogate models used by SAMOTA. For this comparison, the same configuration used for the CD algorithms is considered, training SAMOTA from scratch. The CD algorithm used is GFCI, the one used in CART. Figure 4.2 reports the violin plots. The prediction made by using the causal model via interventions has considerably better (i.e., smaller) %RMSE (5% *vs* 15%), and with much smaller variance (1.8 E-05 *vs* 1.5 E-03), meaning that it yields estimates of the expected test output (hence of the fitness) much closer to the real values and more stable. As a consequence, SAMOTA is expected to generate worse tests (in terms of fitness). The next RQs will investigate this hypothesis. Interestingly, SAMOTA has a slightly better (i.e., bigger) RBO (about 0.68 vs 0.64). This means that SAMOTA surrogates could rank a set of tests slightly better, even though their predicted fitness is far from the real one; this can still be useful for prioritizing existing tests.

4.6.2 RQ2: Coverage of safety requirements

For RQ2, all the testing techniques are run 20 times for 2 hours. For SAMOTA and CART, which use an initial database D, $|D| = 39$ and $|D| = 100$ test scenarios are considered. CART uses GFCI as CD algorithm: in the inference step, the sample size for simulating the intervention is kept as $\eta = 1,000$. The comparison is in terms of *coverage*, namely the proportion of safety requirements violated, computed as:

$$coverage(S) = \frac{|C(S)|}{|R|} \times 100 \tag{4.5}$$

where $S = \{s_1, s_2, \dots s_k\}$ is the testing session executing k tests scenarios, $C(S)$ denotes the set of requirements violated by at least one test case in S, and R is the set of safety requirements under consideration (6 in our case).

To answer RQ2.1 the coverage after the 2-hours testing session is evaluated. Figure 4.3 shows the distribution of the coverage values over 20 repetitions. Except for CART_{100}, all the techniques have a comparable median (white dots) around 33%, namely, all are able to cover 2 out of 6 requirements. None of them covers all requirements. CART and SAMOTA-I cover 4 requirements in 2 and 1 cases respectively, achieving a coverage of 36.67%. With a larger database, CART violates consistently more than 2 requirements: CART_{100} has a median coverage of 50% (33.3% for SAMOTA-I_{100}) and violates 4 requirements (i.e., 66.7%) in 6 repetitions (SAMOTA_{100} violates 4 requirements in 2 repetitions).

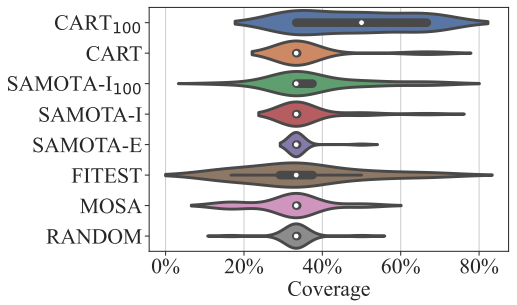


Figure 4.3. RQ2.1 - Coverage effectiveness

Table 4.4. RQ2.1 - Pairwise comparison

vs	CART	SAMOTA-I_{100}	SAMOTA-I	SAMOTA-E	FITEST	MOSA	RANDOM
$CART_{100}$	**7.40E-03**	5.51E-02	**1.65E-02**	**1.40E-03**	**1.00E-03**	**<1.00E-04**	**5.00E-04**
CART	-	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00
SAMOTA-I_{100}	-	-	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00
SAMOTA-I	-	-	-	1.00E+00	1.00E+00	1.00E+00	1.00E+00
SAMOTA-E	-	-	-	-	1.00E+00	1.00E+00	1.00E+00
FITEST	-	-	-	-	-	1.00E+00	1.00E+00
MOSA	-	-	-	-	-	-	1.00E+00

The Friedman test detects a significant difference for at least one pair (p-value = 2.66E-04). The Dunn test [37] is run for *post hoc* analysis (that protects against the multiple comparison problem) to detect which pair of techniques differ significantly. It confirms that $CART_{100}$ is significantly better than all the others (p-values are: 0.0165, 0.0074, 0.0014, 0.0010, 0.0005, and <0.0001 for $CART_{100}$ against, respectively: SAMOTA-I, CART, SAMOTA-E, FITEST, RANDOM, and MOSA) except against SAMOTA-I_{100} but with a p-value slightly above 0.05 (0.0551). In all the other pairs, differences are not significant – the p-values for all the pairs are in the Table 4.4.

Table 4.5 reports, for each technique, the number of repetitions (out of 20) that covered the safety requirement. The two requirements DV and DT are clearly the easiest to cover as all the techniques are able to consistently find the respective violations with at least one scenario per repetition, and, in some cases, even in every repetition ($CART_{100}$, $CART_{39}$, SAMOTA-I_{39}, SAMOTA-E). On the other hand, DCL and DS are more difficult to cover, with techniques covering them in less (or equal for $CART_{100}$ on the latter requirement) than half of the repetitions; TR is covered only once by $CART_{100}$, SAMOTA-E, and MOSA. DP turned out to the hardest to cover in our setting; no technique succeeds in covering it.

The superiority of $CART_{100}$ shows that a bigger dataset has led to higher-quality models and consequently to better test suites. Although a small improvement is noticed also in SAMOTA-I_{100} compared to the other SAMOTA versions, the impact of a bigger archive size is not so pronounced as in CART. This suggests that causal models can benefit more from greater archive size. In Section 4.6.5 this hypothesis is further investigated.

An important advantage of causal models is their interpretability. They

Table 4.5. RQ2.1 - Number of repetitions (out of 20) that violate requirements

Technique	DCL	DV	DP	DS	DT	TR
$CART_{100}$	7	20	0	10	20	1
$CART_{39}$	2	20	0	2	20	0
SAMOTA-I_{100}	5	19	0	2	20	0
SAMOTA-I_{39}	3	20	0	1	20	0
SAMOTA-E	0	20	0	0	20	1
FITEST	5	13	0	2	18	0
MOSA	2	15	0	0	19	1
RANDOM	2	18	0	0	20	0

Legend: DCL: distance from the center of the lane, DV: distance from other vehicles, DP: distance from pedestrians, DS: distance from static obstacles, DT: distance traveled, TR: traffic rules

are human-readable and allow a quick identification of inputs causally related to outputs, and consequently to safety violations. In particular, inspecting the results and looking at the model, some input-output relations can be spotted. For instance, it is possible to note that collisions with other vehicles were mainly caused by the target speed of the ego vehicle, the weather conditions, and the variables specifying the presence of other vehicles (mainly the presence of a vehicle ahead of the ego vehicle in the same lane). When the speed is too high in rainy conditions, the ego vehicle hardly manages to avoid a collision with a vehicle on the same lane that slows down. Similarly, the collisions with pedestrians/static objects were heavily impacted by the roads the ego vehicle was spawned to (e.g., junction, straight road, highway).

More in general, engineers can get different types of insights from a causal model in a relatively simple way compared to ML models. Figure 4.4 reports an excerpt of causal graph obtained, along with the weights from the structural equation coefficients representing how much, on average, the effect is expected to change for a change in the cause.

It is possible to get some insights from the model:
i) from the graph, note that Road ID (i.e., the part of the CARLA's map where the scenario takes place), Vehicle target speed, Road type (e.g.,

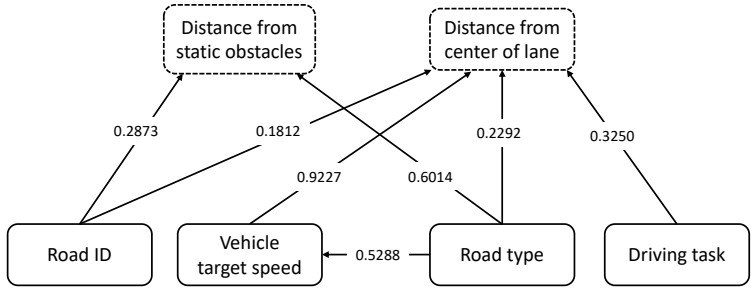


Figure 4.4. Excerpt of causal model

"cross road", "left/right turn", "straight") and Driving task ("follow road", "take 1st/2nd/3rd exit") are all causally related to DCL. Furthermore, it is possible to note that there are important differences between the mentioned causes for the DCL effect. In fact, Vehicle target speed is also impacted by Road type. This means that an even strong correlation between Vehicle target speed and DCL can be well due to Road type that can cause both Vehicle target speed and DCL, namely Road type is a confounder. Any prediction based on that correlation without accounting (i.e.,controlling for) Road type would fail. With causal inference, one can predict the causal effect of Vehicle target speed, net of confounders and correctly attribute the cause for an observed effect.

ii) The graph highlights that the violations of DS and DCL can be caused by multiple variables together, and the underlying equations express the strength of the causal effect of the causes. Also, common causes are also of interest; for instance Road type is a common cause of both DCL and DS.

iii) Several other patterns can be inspected. For instance, the chains of causality connecting multiple inputs can be investigated, and assess the direct and indirect effect (i.e., through other input variables) of an input variable on the output. Other patterns can be identified [141], such as the front-door criterion, highlighting situations in which it is not immediate to derive a causal effect of an input of interest to the output.

The second sub-question of RQ2 (RQ2.2) was about the efficiency in covering safety requirement.s To answer RQ2.2, the coverage over testing time every 20 minutes is measured. Figure 4.5 shows that CART-100

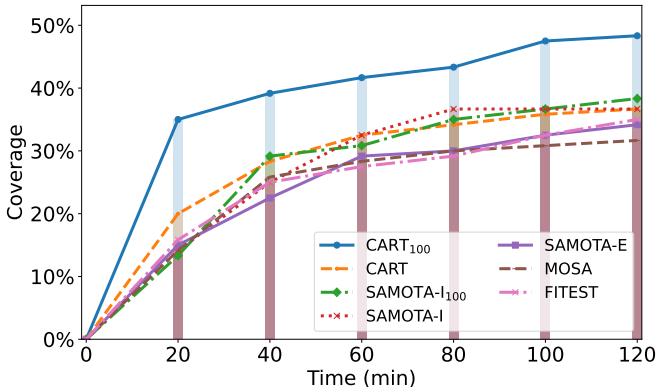


Figure 4.5. RQ2.2 - Coverage efficiency

achieves by far the highest coverage, with a rapid increase in the first 20 minutes. Also CART is better than the baselines in the first 20 minutes. After 40 minutes, CART and SAMOTA-I_{100} are equivalent and slightly better than the other baselines, and keep this superiority along the whole test duration.

RQ2 answer

CART covers a proportion of safety requirements comparable to the other techniques, but more efficiently, as it covers a higher proportion than SAMOTA-I, SAMOTA-E, MOSA and FITEST in the first 40 minutes. When fed with a larger knowledge ($CART_{100}$, using 100 tests to build the model), it markedly outperforms the other techniques, in terms of both proportion of violated requirements and of efficiency.

4.6.3 RQ3: Detection of safety violations

With RQ3, the aim is to evaluate techniques in their ability to expose multiple violation of (possibly the same) safety requirements. Indeed, while RQ2 already shows that CART covers more safety requirements than competitors and earlier, having only one example for a violated safety requirement could be not satisfying for a tester. Multiple diverse violating tests highlight possible different conditions under which a violation occurs, each of which could trigger different faults leading to that violation. This

Table 4.6. Example of multiple safety-violating tests for DV requirement

Road Type	Road ID	Vehicle in Adjacent Lane	Two-wheeled Vehicle in Front	Two-wheeled Vehicle in Adjacent Lane	Two-wheeled Vehicle in Opposite Lane	Weather	Vehicle Target Speed	DV
Cross Road	0	0	1	1	1	Cloudy	40	Violated
Cross Road	1	0	1	0	0	Cloudy	40	Violated
Straight	3	0	1	0	0	Wet/cloudy	30	Violated
Right Turn	2	0	1	1	1	Cloudy	40	Violated
Cross Road	2	0	1	0	1	Cloudy	40	Violated
Cross Road	2	0	1	0	0	Cloudy	40	Violated
Cross Road	2	0	1	0	1	Cloudy	40	Violated
Straight	3	0	1	1	0	Clear	40	Violated
Right Turn	2	0	0	0	0	Wet	40	Violated
Cross Road	1	1	0	1	1	Wet/cloudy	40	Violated
Cross Road	2	0	0	1	0	Wet	40	Not violated
Cross Road	2	0	0	0	0	Wet	40	Not violated

supports the root cause analysis and debugging task.

For instance, let us assume to have a same safety violation of the DV safety requirement exposed by multiple failing tests, as shown in Table 4.6. The Table shows that the DV violation is present in quite different scenarios (with different values of the "Road ID", i.e., the part of the CARLA's map where the scenario takes place, and "Weather" conditions), which can be due to different faults, activated by different combinations of (possibly a subset) of the variables. Except for one test (red cells), the failed tests have at least one input among "Two-wheeled Vehicle in Front", "Two-wheeled Vehicle in Adjacent Lane" and "Two-wheeled Vehicle in Opposite Lane" equal to 1 (yellow cells). This suggests that the presence of a two-wheeled vehicle is a possible fault trigger, hence providing insight into the root cause of the violation - e.g., the autopilot's DNN for obstacle detection could be inaccurate in detecting two-wheeled vehicles. Having multiple violations (i.e., multiple manifestations of the fault) helps to spot the problem. In addition, the Table shows that the violation is present also when no two-wheeled vehicle is present (the red row). The red-cells test has two input values, "Road Type" and "Road ID" that are the same as another failing test (orange cells). Therefore, this can suggest an additional fault, or at least a more complex activation pattern not depending solely on the "two-wheeled" input variables.

If engineers had just one of the above tests, it would be much harder to derive some conclusion about possible causes of the safety violation. The implications are therefore for the root cause analysis and debugging phase.

To assess the ability of the techniques of finding multiple violations, all the techniques are run again 20 times for 2 hours, but with a *fixed* fitness function. The comparison is in terms of number of violations of

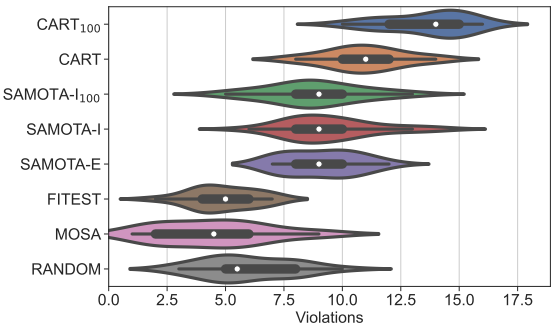


Figure 4.6. RQ3.1 - Number of violations

safety requirements exposed by a testing session S, denoted as $v(S)$. $v_i(S)$ denote the number of violations of the i-th safety requirement.

Results for RQ3.1 (how many violations are found) are in Figure 4.6. The Friedman test detects a significant difference for at least one pair (p-value = 2.2E-16). Table 4.7 reports the p-values for pairwise comparisons, again with the Dunn test. CART_{100} is confirmed to significantly outperform all the algorithms. CART significantly outperforms FITEST, MOSA and RANDOM, while, despite the greater average value, cannot be claimed to be different from all SAMOTA variants.

Figure 4.7 shows the average number of tests violating at least one safety requirement, and the average number of executed tests. Since a test can violate more than one requirement, this is different from the number of violations. Besides exposing more violations, CART_{100} generates more safety-violating tests (as well as a higher ratio of safety-violating tests over

Table 4.7. RQ3.1 - Number of violations, pairwise comparison

vs	CART	SAMOTA-I_{100}	SAMOTA-I	SAMOTA-E	FITEST	MOSA	RANDOM
CART_{100}	5.25E-01	**4.00E-04**	**4.90E-03**	**1.50E-03**	**<1.00E-04**	**<1.00E-04**	**<1.00E-04**
CART	-	2.02E-01	8.18E-01	4.56E-01	**<1.00E-04**	**<1.00E-04**	**<1.00E-04**
SAMOTA-I_{100}	-	-	1.00E+00	1.00E+00	**5.70E-03**	**5.40E-03**	1.12E-01
SAMOTA-I	-	-	-	1.00E+00	**4.00E-04**	**4.00E-04**	**1.33E-02**
SAMOTA-E	-	-	-	-	**1.50E-03**	**1.40E-03**	**3.62E-02**
FITEST	-	-	-	-	-	1.00E+00	1.00E+00
MOSA	-	-	-	-	-	-	1.00E+00

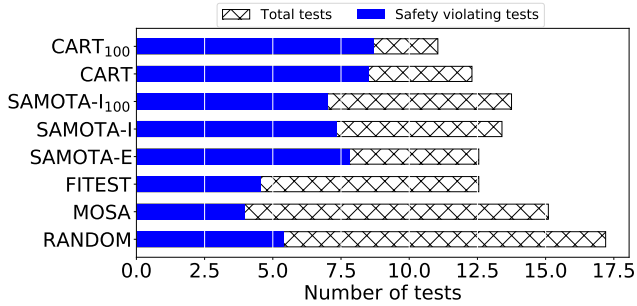


Figure 4.7. RQ3.1 - Number of tests and safety-violating tests

the executed ones).

Figure 4.8 shows the analysis on the ability of tests to violate more than one requirement, representing very critical scenarios. The Figure shows the number of tests violating 1, 2, and 3 safety requirements together (no test violates more than 3 requirements). $CART_{100}$ with the fixed fitness function is the only technique generating more tests that violate two requirements than those violating a single requirement. It also finds a test violating 3 requirements together (DV, DS, TR). CART is the second technique generating tests violating two requirements, followed by SAMOTA-I and SAMOTA-I_{100}.

Table 4.8 reports the average violations of each requirement. The DP and TR requirements are violated by no technique in this fixed-fitness

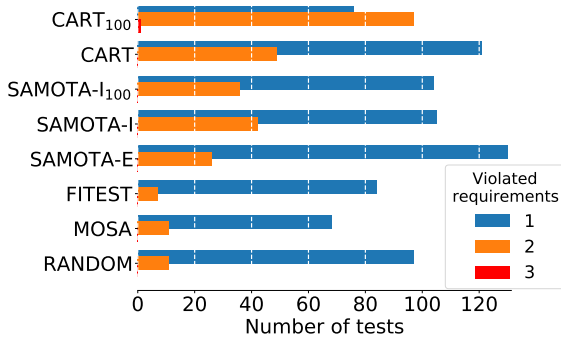


Figure 4.8. RQ3.1 - Number of tests with one or more violations

Table 4.8. RQ3.1 - Average number of violations

	$\overline{q}_i$						$\overline{q}$
Technique	DCL	DV	DP	DS	DT	TR	
$CART_{100}$	0.00	6.10	0.00	0.05	7.50	0.00	13.65
CART	0.05	5.25	0.00	0.00	5.65	0.00	10.95
SAMOTA-I_{100}	0.15	3.20	0.00	0.00	5.45	0.00	8.80
SAMOTA-I	0.05	3.50	0.00	0.00	5.90	0.00	9.45
SAMOTA-E	0.10	2.45	0.00	0.00	6.55	0.00	9.10
FITEST	0.15	1.70	0.00	0.00	3.05	0.00	4.90
MOSA	0.25	2.10	0.00	0.00	2.15	0.00	4.50
RANDOM	0.10	2.65	0.00	0.00	3.20	0.00	5.25

Legend: DCL: distance from the center of the lane, DV: distance from other vehicles, DP: distance from pedestrians, DS: distance from static obstacles, DT: distance traveled, TR: traffic rules

configuration. DT and DV are the most frequently violated, followed by DCL, which is however never violated by $CART_{100}$. Except $CART_{100}$, no other technique violates DS. It is worth noting that, while the adaptive function (used in RQ2) tends, when a requirement is already covered, to target other requirements, the fixed function tends to violate more often some specific, more failure-prone, requirements (e.g., many violations to DV and DT are generated). For instance, in the adaptive case, $CART_{100}$ was able to cover 5 different requirements in 6 out of 20 repetitions, which never happened in this configuration, but of course detecting a lower average number of total violations (9.4 *vs* 13.65).

Finally, Figure 4.9 plots the efficiency comparison (RQ3.2). It clearly shows that both $CART_{100}$ and CART detect more safety violations since the beginning, thus supporting the early identification of critical scenarios.

RQ3 answer

$CART_{100}$ and CART detect the greatest number of violations after the entire testing session, expose more violations earlier, generate more safety-violating tests and with a better ratio of safety-violating tests over total tests.

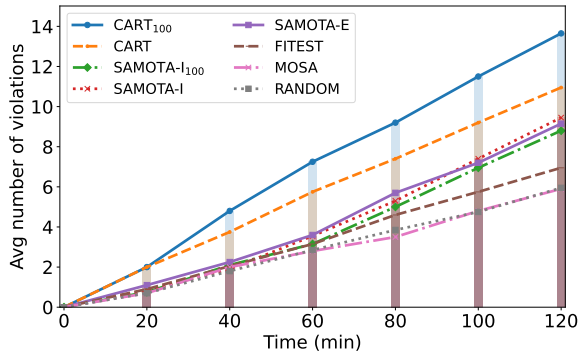


Figure 4.9. RQ3.2 - Efficiency

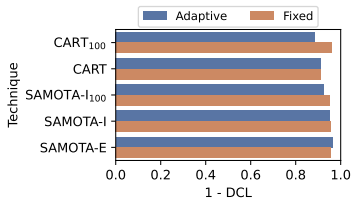
4.6.4 RQ4: fitness and diversity

RQ4 investigates the *fitness* values and *diversity* of the generated tests. This comparison is done on the test suites produced in RQ2 and RQ3 (with the adaptive and fixed fitness functions, respectively) by the best-performing techniques, which are: CART_{100}, CART, SAMOTA-I_{100}, SAMOTA-I and SAMOTA-E.

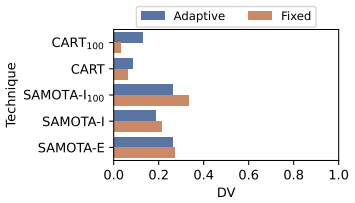
Fitness evaluation

To evaluate the fitness of tests, the values of the output variables of each test are considered (i.e., DT, DP, DV, DS, DCL). The TR output is excluded, since it is not a continuous but a binary variable (all other 5 are distances).

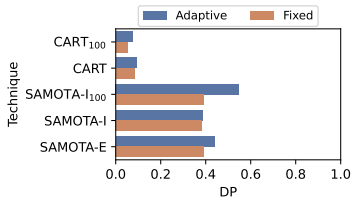
Figures 4.10 and 4.11 report the (min-max) normalized values for each output variable, averaged over all the test scenarios generated in the 20 repetitions, split in two groups: failing (i.e., safety-violating) and non-failing tests. For all of them but DCL, the smaller the better; for DCL, (1−DCL) is reported. The plots show that CART and CART_{100} achieve better values for almost all the 10 output variables (5 for the *adaptive* fitness function, RQ2, and 5 for the *fixed* one, RQ3), except for DT, in both the datasets. The results are confirmed for both groups, with, expectedly, a more pronounced advantage of CART in failing tests.



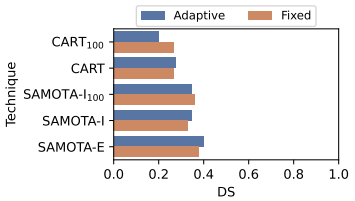
(a) Distance from center of lane



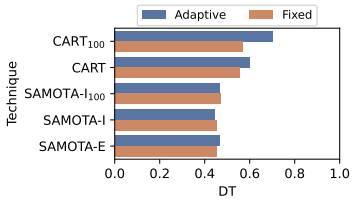
(b) Distance with other vehicles



(c) Distance with pedestrians

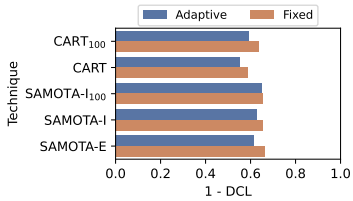


(d) Distance with static obstacles

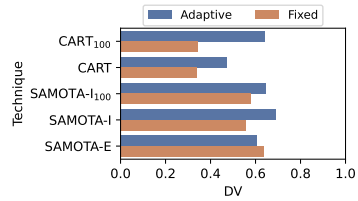


(e) Distance traveled

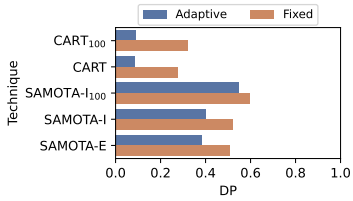
Figure 4.10. RQ4 - test suite fitness per requirement (failing tests)



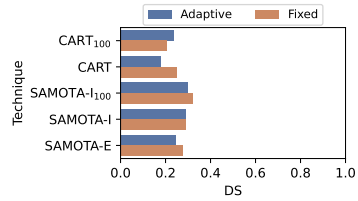
(a) Distance from center of lane



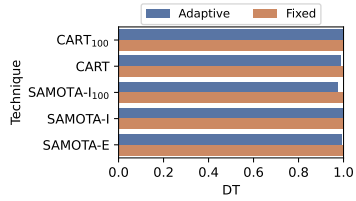
(b) Distance with other vehicles



(c) Distance with pedestrians



(d) Distance with static obstacles



(e) Distance traveled

Figure 4.11. RQ4 - test suite fitness per requirement (not failing tests)

Considering the whole test set, the Friedman hypothesis test rejects the null hypothesis of no difference between the techniques (p-value $<$ 0.05) in all the cases but two: DT and DV in the *adaptive* case (p-values: 0.949, 0.9578, respectively). Table 4.9 reports the p-values for the pairwise comparison with the Dunn test for the *adaptive* case. The comparison highlights that CART and CART_{100} are significantly better (p-value $<$ 0.05) than all SAMOTA variants for three outputs (DP, DCL, DS). While the difference is confirmed to be not significant for DT and DV (p-values for DV are not reported as they were all $1.00E+00$).

In the *fixed* case, the Friedman test rejects the null hypothesis of no

Table 4.9. RQ4 - p-values for pairwise comparison. Adaptive fitness

(a) Distance from the Center of the Lane (DCL)

vs	CART	SAMOTA-I_{100}	SAMOTA-I	SAMOTA-E
$CART_{100}$	1.00E+00	**1.14E-02**	**<1.00E-04**	**1.00E-04**
CART	-	**2.50E-03**	**<1.00E-04**	**<1.00E-04**
SAMOTA-I_{100}	-	-	1.00E+00	1.00E+00
SAMOTA-I	-	-	-	1.00E+00

(b) Distance with Pedestrians (DP)

vs	CART	SAMOTA-I_{100}	SAMOTA-I	SAMOTA-E
$CART_{100}$	4.00E-01	**<1.00E-04**	**1.68E-02**	**3.20E-03**
CART	-	**<1.00E-04**	**<1.00E-04**	**<1.00E-04**
SAMOTA-I_{100}	-	-	1.00E+00	1.00E+00
SAMOTA-I	-	-	-	1.00E+00

(c) Distance with Static obstacles (DS)

vs	CART	SAMOTA-I_{100}	SAMOTA-I	SAMOTA-E
$CART_{100}$	1.00E+00	**1.03E-02**	**5.00E-04**	**5.00E-04**
CART	-	**1.00E-04**	**<1.00E-04**	**<1.00E-04**
SAMOTA-I_{100}	-	-	1.00E+00	1.00E+00
SAMOTA-I	-	-	-	1.00E+00

(d) Distance Traveled (DT)

vs	CART	SAMOTA-I_{100}	SAMOTA-I	SAMOTA-E
$CART_{100}$	1.00E+00	1.00E+00	6.20E-02	2.85E-01
CART	-	1.00E+00	6.20E-02	2.85E-01
SAMOTA-I_{100}	-	-	1.00E+00	1.00E+00
SAMOTA-I	-	-	-	1.00E+00

difference between in all the cases. Table 4.10 reports the p-values for the pairwise comparison. In this case, $CART_{100}$ and CART are significantly better than SAMOTA in the DP, DV and DS output (with the exception of SAMOTA-I *vs* $CART_{100}$ for DS, p-value = 0.1435). For DCL, the significant differences are CART *vs* SAMOTA-I and CART *vs* SAMOTA-E (the others, although in favour of CART, are not significant). For DT, the only significant difference is SAMOTA-I_{100} *vs* $CART_{100}$, in favour of the former.

Overall, the results indicate that $CART_{100}$ and CART generate tests that push almost all the output variables closer to the thresholds, generating near-violating scenarios, especially with the *fixed* fitness. These could better highlight critical behaviours (e.g., the car stops very close to a pedestrian, which might be not desirable even though not a collision), and can make it easier to manually derive violating scenarios by tuning the obtained near-critical ones. Moreover, since CART follows an evolutionary approach, these high-fitness tests would more likely evolve into violating tests with more testing time available for the algorithm.

Diversity evaluation

As for diversity of the test suites, the Test Set Diameter (TSD) is used, a well-known metric for black-box test suite diversity used in test prioritization [43, 120, 62], computed on the 20 repetitions for both test suites (generated in RQ2 and RQ3). The TSD is computed via the Normalized

Table 4.10. RQ4 - p-values for pairwise comparison. Fixed fitness

(a) Distance from the Center of the Lane (DCL)

vs	CART	SAMOTA-I_{100}	SAMOTA-I	SAMOTA-E
$CART_{100}$	9.20E-02	1.00E+00	1.00E+00	6.23E-01
CART	-	7.40E-02	**2.00E-03**	**<1.00E-04**
SAMOTA-I_{100}	-	-	1.00E+00	7.34E-01
SAMOTA-I	-	-	-	1.00E+00

(b) Distance with other Vehicles (DV)

vs	CART	SAMOTA-I_{100}	SAMOTA-I	SAMOTA-E
$CART_{100}$	1.00E+00	**<1.00E-04**	**<1.00E-04**	**<1.00E-04**
CART	-	**<1.00E-04**	**1.00E-03**	**1.00E-04**
SAMOTA-I_{100}	-	-	1.00E+00	1.00E+00
SAMOTA-I	-	-	-	1.00E+00

(c) Distance with Pedestrians (DP)

vs	CART	SAMOTA-I_{100}	SAMOTA-I	SAMOTA-E
$CART_{100}$	1.00E+00	**<1.00E-04**	**<1.00E-04**	**<1.00E-04**
CART	-	**<1.00E-04**	**<1.00E-04**	**<1.00E-04**
SAMOTA-I_{100}	-	-	1.00E+00	1.00E+00
SAMOTA-I	-	-	-	1.00E+00

(d) Distance with Static obstacles (DS)

vs	CART	SAMOTA-I_{100}	SAMOTA-I	SAMOTA-E
$CART_{100}$	3.77E-01	**4.50E-02**	1.43E-01	**5.50E-03**
CART	-	**<1.00E-04**	**<1.00E-04**	**<1.00E-04**
SAMOTA-I_{100}	-	-	1.00E+00	1.00E+00
SAMOTA-I	-	-	-	1.00E+00

(e) Distance Traveled (DT)

vs	CART	SAMOTA-I_{100}	SAMOTA-I	SAMOTA-E
$CART_{100}$	5.85E-02	**3.17E-02**	1.00E+00	1.00E+00
CART	-	1.00E+00	3.54E-01	1.41E-01
SAMOTA-I_{100}	-	-	2.16E-01	8.08E-02
SAMOTA-I	-	-	-	1.00E+00

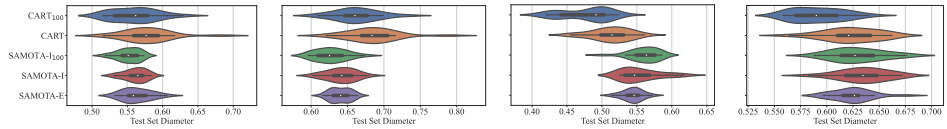
Compression Distance (NCD_1):

$$NCD_1(X) = \frac{C(X) - min_{x \in X}\{C(x)\}}{max_{x \in X}\{C(X \backslash \{x\})\}} \tag{4.6}$$

$$TSD(X) = max\{NCD_1(X), max_{Y \subset X}\{TSD(Y)\}\} \tag{4.7}$$

where $x \in X$ denotes a test, X the whole test suite, and $C(X)$ is the length of compressing X with the *bzip2* compression program. [43]
Figure 4.12 shows that CART with the adaptive fitness function produces test suites with more diverse inputs and outputs. For the inputs (Fig. 4.12a), the difference is however significant only for CART *vs* SAMOTA-I_{100} (p-value: 0.0157), while for the outputs (Fig. 4.12b) the p-values are: 0.0013, <0.0001, and 0.0006 *vs*, respectively, SAMOTA-I, SAMOTA-I_{100}, SAMOTA-E). The second one is $CART_{100}$. With the *fixed* fitness, the test suites, especially those by $CART_{100}$, have (statistically) worse diversity than SAMOTA in both inputs and outputs (Fig. 4.12c, 4.12d), while CART is statistically equivalent to SAMOTA for outputs and worse for inputs. Tables 4.11 and 4.12 report the p-values for all the pairwise comparisons.

The observed smaller diversity in the fixed fitness function is a side



(a) adaptive, input **(b)** adaptive, output **(c)** fixed, input **(d)** fixed, output

Figure 4.12. RQ4 - test suite diversity

effect: the focus of the fixed fitness function on exposing critical tests is exploited by the CART's causal models more than the SAMOTA's surrogate models to learn input-output relations, giving higher fitness but also more similar tests. Improving diversity with the fixed function too is an interesting future step worth investigating.

RQ4 answer

With the adaptive fitness function, CART and $CART_{100}$ produce test suites that have higher fitness values for 4 out of 5 requirements and have a better diversity than the compared algorithms. With the fixed fitness function, they produce test suites with (even) higher fitness values for 4 out of 5 requirements, but with worse diversity.

4.6.5 Additional remarks

Like SAMOTA-I, CART requires an initial database of executed scenarios to build the model, which can be taken from historical driving data or from past tests data. When unavailable, the database generation entails a start-up cost. This is, however, paid off not only by a more effective and efficient testing, but also by yielding a causal model characterising the sys-

Table 4.11. RQ4 - p-values for pairwise comparison. Adaptive fitness

(a) TSD on input

vs	CART	$SAMOTA\text{-}I_{100}$	SAMOTA-I	SAMOTA-E
$CART_{100}$	5.54E-01	1.00E+00	1.00E+00	1.00E+00
CART	-	**1.57E-02**	1.00E+00	1.00E+00
$SAMOTA\text{-}I_{100}$	-	-	1.00E+00	1.00E+00
SAMOTA-I	-	-	-	1.00E+00

(b) TSD on output

vs	CART	$SAMOTA\text{-}I_{100}$	SAMOTA-I	SAMOTA-E
$CART_{100}$	9.01E-01	**1.10E-03**	3.38E-01	2.01E-01
CART	-	**<1.00E-04**	**1.30E-03**	**6.00E-04**
$SAMOTA\text{-}I_{100}$	-	-	8.11E-01	1.00E+00
SAMOTA-I	-	-	-	1.00E+00

Table 4.12. RQ4 - p-values for pairwise comparison. Fixed fitness

(a) TSD on input

vs	CART	SAMOTA-I_{100}	SAMOTA-I	SAMOTA-E
$CART_{100}$	4.34E-01	**<1.00E-04**	**<1.00E-04**	**<1.00E-04**
CART	-	**<1.00E-04**	**2.60E-03**	**2.62E-02**
SAMOTA-I_{100}	-	-	1.00E+00	9.21E-01
SAMOTA-I	-	-	-	1.00E+00

(b) TSD on output

vs	CART	SAMOTA-I_{100}	SAMOTA-I	SAMOTA-E
$CART_{100}$	**9.70E-03**	**5.00E-04**	**<1.00E-04**	**4.00E-03**
CART	-	1.00E+00	1.00E+00	1.00E+00
SAMOTA-I_{100}	-	-	1.00E+00	1.00E+00
SAMOTA-I	-	-	-	1.00E+00

Table 4.13. Coverage of safety requirements of CART with different initial database size

Technique	Mean	Std.Dev
$CART_{150}$	0.48	0.06
$CART_{100}$	0.48	0.14
CART	0.37	0.10

tem under test – a useful asset to support other quality-related activities besides testing. Also, with a causal model, it is relatively easy to embed domain knowledge by engineers, which would further boost performance. Specifically, adding knowledge requires specifying known (or forbidden) cause-effect edges in the graph. In the evaluation, this option has been left out for the sake of full automation, but also to avoid human intervention that could have biased the result (in favour of CART). Indeed, in a real setting known cause-effect relations would be added to the model so as to fine-tune CART for a certain context.

Related to this, the results suggest that having a larger database could be beneficial. To assess this hypothesis, an additional experiment has been performed with CART, setting an initial dataset of 150 random tests, 20 repetitions. Results are reported in Table 4.13. They show that both $CART_{100}$ and $CART_{150}$ improve over CART with 39 initial tests. On the other hand, no significant difference can be noticed between $CART_{150}$ and $CART_{100}$, except a better standard deviation for the former – hence more stable results. By inspecting the causal graph generated by the causal structure discovery algorithms in the two cases, they turned out to be the same. It is possible to conclude that 100 test cases were enough in this scenario to learn a close approximation of the "true" causal model, and further tests add little.

Finally, the CI queries to derive a test impact the test generation time.

The impact is however negligible, as the average generation time for a test case over all the CART (and CART_{100}) executions turned out to be 7.25 seconds per test, which is 1.3% of the average test execution time (that is $\approx$ 9 minutes). As shown by the efficiency plots (Fig. 4.5 and 4.9), this has not undermined the gain of our solution. If needed, this time can be reduced by implementing policies for sampling values for the intervention (i.e., for the CI queries). The literature proposes some strategies in this regard. For instance, an option is to select the intervention maximizing the information gained (i.e., minimize the uncertainty) about the true graph – hence intervene to better learn the "true" graph [169]. A different path can be to set an intervention trying to maximize/minimize the desired test objective(s) (e.g., increase coverage or fitness), or maximize diversity. These criteria can be implemented in several ways, e.g., via probabilistic sampling (e.g., (non-)uniform random sampling), search- or learning-based techniques, adaptive strategies (using previous selections to drive the next ones).

4.7 Threats to validity

The time budget for each technique is set at 120 minutes (according to [55]). This is both to limit the experimentation (almost 700 computing hours) and to fairly compare with previous experiments. Giving more or less time to each technique could lead to different results. Even if the input variables (16 input variables) constitute a large input space, it can be worth considering more variables in a real-world system. This could affect the performance of causal discovery, for which scalability is an open challenge. However, CSD algorithms have shown tractable running time on datasets with up to 500 variables [148], which is indeed a remarkable upper bound for setting up effective testing sessions.

The use of a simulator can be a further threat to validity. For instance, it may rarely generate scenarios physically impossible. However, manual inspection of the executed scenarios did not reveal any such case. The use of simulators is widespread in ADS, as testing in the real world is costly, and this has pushed toward the development of high-fidelity simulators like CARLA. Indeed, challenges are still open about simulators' representativeness [170], but they are increasingly trustable and are often

an obliged path [57], [58].

The initial database of tests provided to CART and SAMOTA-I can affect performance; two databases have been used, one provided by SAMOTA's authors and one of the randomly generated tests, to mitigate the impact. Replicating with different databases would enforce the results. Also, the results are obtained under default setting of `Tetrad` and `DoWhy`; a fine-tuning of CSD algorithms and CI estimation methods is left to future work.

Despite the efforts to ensure defect-free code – the CART prototype and the infrastructure code interfacing to CARLA, partially borrowed from [55] - their presence cannot be excluded. Last but not least, experiments are on one specific ADS and simulator. Although Pylot and CARLA are representative widely-used choices [131, 39], replicating the experiments with other subjects is needed to improve external validity. Finally, CART is evaluated on the same six safety requirements used by SAMOTA; however, CART is not tied to these specific requirements, it can be used with any other safety requirement of interest, provided that their violation can be checked (i.e., there is an oracle).

Chapter 5

Instantiating RBST for stateful testing

5.1 Preliminaries

In many environments, it can be beneficial to exploit dynamic factors and formulate the testing tasks in a *stateful* manner, where a test case consists of a sequence of inputs. This approach can be particularly useful when testing dynamic systems that evolve over time in response to input sequences. When the dynamic behavior of the SUT is described through control- or data-flow graphs, various model-based testing techniques can be applied (Section 2.2.2). These techniques, by exploring diverse input sequences, are especially effective for systems with smaller, more controlled state spaces.

On the other hand, in complex, black-box environments with large, uncertain state and input spaces — such as in online ADS testing — it can be challenging to predetermine the effects of specific input sequences on the SUT. Here, RL is often employed to automate testing tasks: by allowing an RL agent to interact with the environment using a well-defined objective function, it can autonomously learn sequences of inputs that are likely to lead to failures. RL has an inherently causal aspect, as it learns a policy for selecting actions — i.e., inputs — by *actively intervening* in the environment. In other words, RL learns from *interventional* data and not *observational* data. In the literature, the connections and synergies

between CR (i.e., with causal frameworks and CI, starting from *observational* data) and RL have been exploited, giving rise to Causal Reinforcement Learning (CRL). In CRL, actual physical interventions done in an environment are combined with causal modeling where interventions are simulated on a causal model of the environment. Its exploitation for software testing can be helpful in dealing with sequential tasks (e.g., *stateful* testing). As discussed in the following of this Chapter, the same synergies can be employed to instantiate RBST for CRL.

5.1.1 Reinforcement Learning

Reinforcement Learning (RL) is the process of learning what to do (i.e., how to relate circumstances to actions) in order to maximize a reward [174]. The *learner* (a.k.a. *agent*) is not instructed on actions to take, but, interacting with its *environment*, explores which actions produce the highest reward.

The RL process can be formalized through a Markov Decision Process (MDP), a classical model for sequential decision-making, where actions do not influence just immediate reward, but also the following states. An MDP is defined by a tuple (S, A, P, R, γ), where: S and A are respectively the sets of states and actions; P is the state transition probability function $P(s_{t+1}|s_t, a_t)$, assigning the probability of state s_{t+1} at time step $t+1$, given state s_t and action a_t at time step t; the reward function $R : S \times A \to \mathbb{R}$ maps a state-action pair to the set of real values; $\gamma \in [0, 1]$ is the discount factor controlling the trade-off between future and immediate rewards.

At time step t, the agent observes the state of the environment s_t, and selects an action a_t based on its *policy* π. The policy is generally a stochastic function $\pi : S \to A$ that yields the probability of selecting action $a_t \in A$ in state $s_t \in S$ at step t. At step $t+1$ the environment outputs the next state s_{t+1} and a scalar value r_{t+1}, rewarding the goodness of action a_t. The reward is the learning signal, that the agent aims to maximize. Through the interactions with the environment, the agent learns an *optimal policy* π^*, that maximizes the total expected reward the agent gets in its lifetime (the expectation accounts for the randomness of both the transition probability function of the environment and the policy). The RL methods in the literature differ in how they update the agent's policy as further experience becomes available.

The most common RL algorithms are *model-free* (not equipped with a model of the environment). Within this family, RL algorithms can be categorized into *value-based*, *policy-based*, or a combination of the two. Value-based algorithms learn a *value function* giving an estimate of "how promising" a state (or a state-action pair) is. The estimate is computed as the total expected reward through a *state-value function* $V(s)$ (or an *action-value function* $Q(s,a)$). The policy is then built by choosing in each state the action that maximizes the value function. Policy-based methods (e.g., policy-gradient) maximize the total expected reward by finding a policy through stochastic gradient ascent with respect to the policy parameters (e.g., the parameters of a neural network).

One of the most important value-based algorithms is *Q-learning*, proposed in 1989 by Watkins [189]. It learns an action-value function $Q(s,a)$, updated as new data becomes available. The agent's knowledge is represented as a table (named *Q-table*) mapping states and actions to the expected reward. At each time step, the agent starts from state s_t, selects the action with the highest Q-value ($\underset{a \in A}{max} Q(s_t, a)$), enters next state s_{t+1}, and collects the reward r_{t+1}. Finally, it updates the Q-value of the starting state-action pair (s_t, a_t) as:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \underset{a \in A}{max} Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

where: γ is the discount factor and $\alpha \in [0, 1]$ is the learning rate, controlling the step size at which the Q-values are updated.

Advances in deep learning have led to the development of deep RL (DRL) algorithms like *Deep Q-Network* (DQN) [121]. DQN combines the *Q-learning* paradigm with a neural network that receives a state as input and approximates the Q-values for each potential action as output. The neural network replaces the Q-table and concisely stores the agent's experience, handling large state spaces such as continuous ones. To enhance stability during training, DQN typically utilizes a buffer of past experiences. During training, it randomly samples batches of experiences to update the weights of the neural network. Additionally, for stabilization, DQN uses an auxiliary network (also called target network), a copy of the network being trained. The weights of such network are kept frozen for a certain number of training steps so that the original network is trained

with a fixed target. DQN handles continuous state spaces, but still requires actions to be discrete, as its update rule requires a maximization over all the actions for a particular state.

Both Q-learning and DQN use a behavior policy in training to explore the environment and look for actions that lead to high rewards. A common policy is ϵ-*greedy*, the parameter ϵ representing the probability of choosing a random action instead of relying on the Q-value function (respectively a table and a neural network). Typically, ϵ is set to 1 at the beginning of training, so the agent starts choosing a random action. As training progresses and the agent acquires knowledge of the environment, ϵ gradually decreases, and the agent starts selecting actions greedily with higher probability. The annealing schedule, i.e., the rate at which ϵ decreases over time, is problem-specific, as it depends on the rate of exploration that is required to effectively learn a task.

5.1.2 Model-based RL

RL systems may use a *model* of the environment to infer about its behavior and, for instance, predict the next state (and reward) given the current one and an action. The use of the model opens the possibility for planning activities (in addition to *learning*), namely deciding the actions to perform by considering possible future circumstances without actually experiencing them. Methods that integrate learning and planning are called *model-based* and differ from *model-free* ones that are explicitly trial-and-error.

Figure 5.1 shows an overview of possible connections between planning and learning. The base functioning of an RL agent includes only steps ⓕ (i.e., use the observations to update the Policy/Value function) and ⓔ (i.e., use the Policy/value function to decide the action, given a certain state). On the other hand, the design of model-based methods can be decomposed into two steps additional steps [122]: the first is *dynamic model learning*, namely learning a model of the environment; the second is *planning* over the model to recommend an action and/or improve a learned policy or value function. Specifically, the step ⓖ includes the training of a model based on real agent experience (i.e., observations) and then use the model for the planning phase ⓐ. The planning phase could consist, for instance, in using the model as a "surrogate" of the environment, to

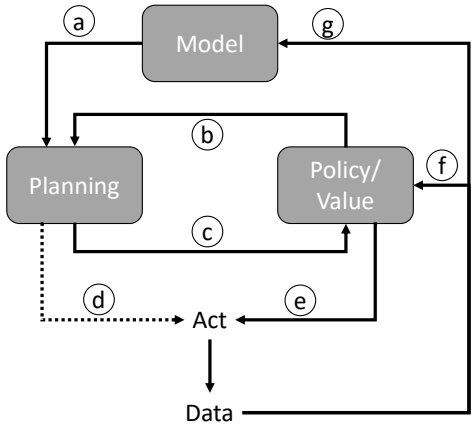


Figure 5.1. Model-based RL overview [122].

generate hypothetical trajectories (or sequences) of state-action pairs. The outcome of the planning phase can be used to directly decide the action ⓓ; here, a number of strategies to combine the decisions from the Policy/Value function and the planning phase can be set up. On the other hand, the planning phase can be used with respect to the Policy/Value function. Specifically, in step ⓒ the outcome of the planning is used as training targets for the policy/value, while in step ⓑ the information from the policy/value is used to improve the planning procedure (e.g., by generating trajectories based on the actual policy/value function).

It is worth stressing that subsets of these steps, and not necessarily all, can be employed to formulate a model-based RL strategy. For example, the model can be learned from experience but, in some environments like the games Go or chess, it can be fully predefined at the beginning (i.e., step ⓖ is not needed). Another example of integrated planning and learning has been presented by Sutton [173] with an architecture called *Dyna*. Here the model is employed to make one-step predictions $S_{t,a} \rightarrow S_{t+1}$ to update the policy (i.e., step ⓒ).

5.2 RBST for Causal Reinforcement Learning

Causal Reinforcement Learning. RL is inherently causal. By design, it is an interactive learning process where the agent learns from data generated through experimentation, or more specifically, by *intervening* in the environment. This interaction can be modeled as a decision-making process using a MDP. The actions taken by the agent directly correspond to *do*-operations in causal inference [47], where each action leads to a specific effect. Therefore, every action taken, given the current state, has a causal relationship not only with the reward but also with the subsequent state.

With the advancements in the CR literature, the connections between RL and causality have been exploited, giving rise to CRL. It can be defined as "a suite of algorithms which aim at embedding causal knowledge into RL for more efficient and effective model learning, policy evaluation, or policy optimization" [200]. It is formalized through two main components: *i)* an RL model setting, e.g., MDP, POMDP, MAB, etc., and *ii)* causal-based information (e.g., an SCM) regarding an environment or task.

Although there is no work so far that applies CRL in software testing, there are works studying causal RL in the model-based RL settings [200, 33]. For instance, Lu *et al.* [107] propose a model-based RL method that estimates an SCM from observational data with the time-invariant confounder between the action and reward based on the latent-variable model proposed by Louizos *et al.* [106]. Gasse *et al.* [47] combine interventional data with observational data to estimate the latent-based transition model, which can be used for model-based RL. Wang et al. [188] focus on how to improve the sample efficiency of the online algorithm by incorporating large amounts of observational data in the model-free RL settings. They follow the assumption that the transition kernels and reward functions are linear so that the corresponding SCMs are also linear [143]. In general, in an online tutorial[1], Bareinboim divides the CRL activities into six tasks: 1) Generalized policy learning, how to learn policy by systematically combining offline and online modes of interaction [201, 128]; 2) deciding when and where to intervene, to refine the policy space [203, 96]; 3) Counterfactual decision-making, defining optimization criterion based

[1] `https://crl.causalai.net`

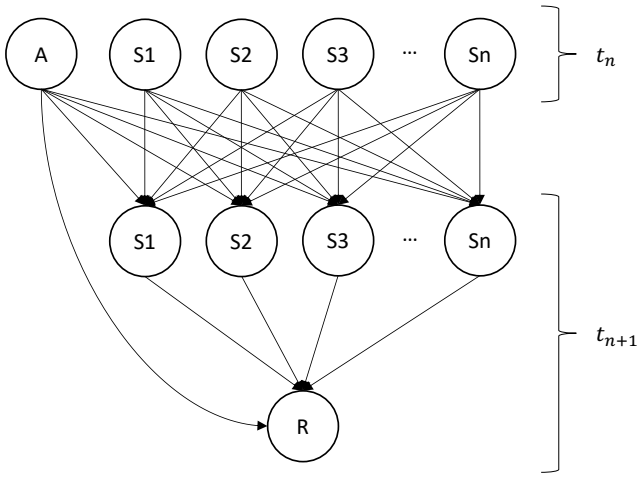


Figure 5.2. MDP causal model design.

on counterfactuals rather than interventions [202]; 4) Generalizability and robustness of causal claims, to study the transportability of policies [26]; 5) Learning causal models, how to discover the causal structure combining both observational and interventional experience [77]; and 6) Causal Imitation Learning, policy learning with unobserved rewards [90]. An important research effort is required in a better exploitation of the advances in the CRL literature, as several of the techniques developed for the mentioned tasks find large application in the software testing domain.

RBST instatiation An MDP can be naturally mapped onto a causal model, as depicted in Figure 5.2. Specifically, the model captures a single transition from time step t_n to t_{n+1}, where each state variable $S_1, \ldots, S_n$ at t_n potentially influences all state variables at t_{n+1}. Although in certain cases, the relationships among state variables might be more constrained, this model assumes no prior knowledge of such dependencies. Therefore, it allows for any variable at t_n to affect all others at t_{n+1}. The action taken at time step t_n affects the subsequent state $S_{t_{n+1}}$ and the reward, and in turn, the state at t_{n+1} influences the reward.

Under this setting, CRL extends traditional model-based RL by integrating causal models that distinguish between observed transitions, counterfactual transitions, and potential transitions. For example, CRL can

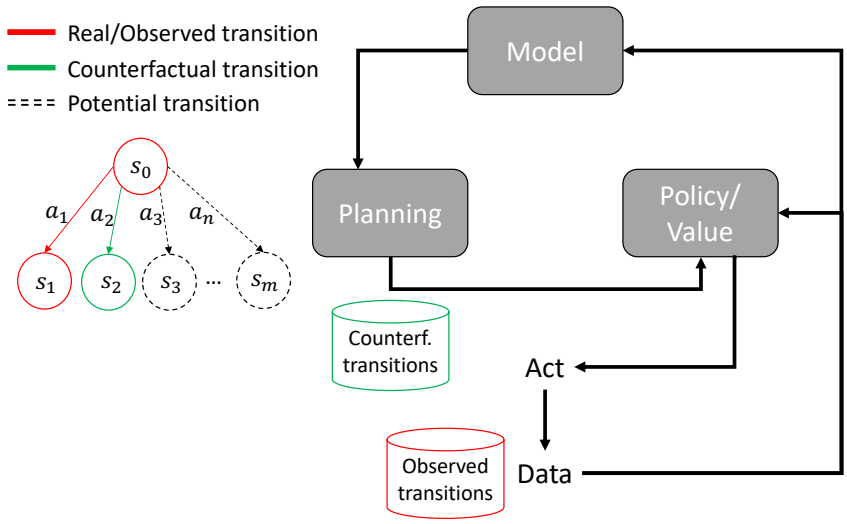


Figure 5.3. Model-based CRL.

leverage interventions and counterfactual reasoning to generate unobserved transitions, which can be used to update the policy of the RL agent. This allows the agent to learn not only from direct experience but also by reasoning about hypothetical scenarios — e.g., what would have happened had a different action been taken in a given state.

Figure 5.3 illustrates this process using the *Dyna* architecture [173]. In this setup, the causal model is used to complement the standard RL components — policy/value functions, actions, and data collection — by adding a causal "Model" and a "Planning" phase. Real experience, represented as transitions of the form $\langle state_{t_n}, action, state_{t_{n+1}}, reward \rangle$, is used to fit a SCM such as the one in Figure 5.2. Then, starting from observed transitions such as $\langle s_0, a_1, s_1, r_1 \rangle$, the causal model is used to infer a number of potential transitions $\langle s_0, a_2, s_2, r_2 \rangle$, representing hypothetical outcomes in case a different action had been taken. Counterfactuals on the SCM allow the agent to explore these unobserved, hypothetical transitions. This enables the agent to plan more effectively by simulating potential outcomes based on hypothetical interventions, thus leveraging both observed and inferred data to improve decision-making.

Although the mentioned process is potentially applicable to every RL

Algorithm 2 DQN with CRL

```
1: Initialize Replay Buffer ℬ, Planning Buffer 𝒫, and DQN agent
2: while not termination condition do
3:     Initialize environment and set initial state s_0
4:     while episode not done do
5:         Select action a_t based on ϵ-greedy policy from Q(s_t, a_t)
6:         Execute action a_t, observe reward r_t and next state s_{t+1}
7:         Store transition t_t = (s_t, a_t, r_t, s_{t+1}) in ℬ
8:         if ℬ size > mini-batch size then
9:             Sample a mini-batch from ℬ and train DQN
10:        end if
11:        if 𝒫 size > mini-batch size then
12:            Sample a mini-batch from 𝒫 and train DQN
13:        end if
14:        if number of steps ≥ target update interval then
15:            Update target policy
16:        end if
17:    end while
18:    Planning Phase:
19:    Select observed transitions T from ℬ
20:    for each transition t in T do
21:        Compute counterfactual transition t_c
22:        Store t_c in 𝒫
23:    end for
24: end while
```

agent, let DQN be the guiding example to explain the functioning. As explained in Section 5.1.1, DQN uses a buffer of past experience (i.e., observed transitions in Figure 5.3) from which, at each timestep, it samples a mini-batch and uses it to train the neural network that approximates the Q-value function. With CRL, this buffer is complemented by a buffer of counterfactual transitions, so that the agent can sample and train from both. More formally, Algorithm 2 illustrates this basic functioning, where the lines due to CRL are highlighted in green. The major addition is the planning phase (lines 18-23). In particular, it requires the selection of a number of transitions from $\mathscr{B}$ (e.g., a predefined number or all the transitions of the last episode) and then for each of them, with the use of the SCM, a counterfactual is computed. In this case, the RBST user is required to select the rationale of the counterfactual. Specifically, both the variable(s) on which to intervene and also the value to assign have to be

chosen. For instance, testers may want to ask what would have happened if a different action was performed in the same state, or the same action was performed but in a different initial state. In any case, the results are stored in $\mathscr{P}$, from which a mini-batch is sampled after each step to train the DQN agent (lines 11-13).

By integrating causal reasoning with model-based planning, CRL can produce more robust policy updates, as it can predict the effects of unseen actions and handle uncertainty in environments where actions exhibit complex interdependencies. Furthermore, this integration holds significant potential for improving the efficiency of the learning process. By reasoning about both observed and hypothetical transitions, especially in complex environments, an agent can significantly boost the exploration of the transition space, thus speeding up the learning.

5.2.1 Motivation study

To validate and evaluate the proposal, the CartPole environment is used as a case study [91], a simple, well-known, RL environment. In this environment, a pole is fixed at one end on top of a moving cart, and the goal is to prevent the pole from falling over by controlling the cart's movement. The state space is a set of four continuous variables representing the position and velocity of the cart, as well as the angle and angular velocity of the pole. The action space is discrete, allowing two possible actions: moving the cart left or right. The reward function is simple — each time step that the pole remains upright, a reward of $+1$ is given, encouraging the agent to balance the pole for as long as possible. The episode ends when the pole falls beyond a certain angle or the cart moves out of bounds, making it a standard problem for testing control algorithms in RL.

For the experiments, DQN is used as it is a more powerful RL agent compared to Q-learning, and inherently handles continuous state spaces. Conversely, to prevent state explosion usually other agents, like simple Q-learning, require "binning" (i.e., discretization of the state space) which increases the complexity of the problem by introducing additional hyper-parameters. DQN is compared against CRL (the model-based version of DQN utilizing CR) and a RANDOM baseline. Specifically, CRL represents the RBST instance depicted in Table 5.2: the causal model is defined by prior/domain knowledge (i.e., the mapping with the MDP, Figure 5.2), the

Table 5.1. Motivation study hyperparameters

Parameter	Value
Learning Rate	0.001
Discount	0.99
Batch size	32
Buffer size	10,000
Plan Buffer size (CRL)	10,000
Target update interval	50

intervention variable is fixed to the action as the aim is to compute counterfactual transitions with different actions, the value is randomly chosen among all possible actions not observed for a given state, and the effect estimation is simulation-based. The hyperparameters used in the experiments are listed in Table 5.1. Since the cartpole environment has only two possible actions, the counterfactual action is limited to the one not taken in the actual observed transition. An ϵ-greedy policy is employed with an exploration budget of 250 episodes, where ϵ linearly decreases from 1 (purely random actions) to 0.1. The maximum cumulative reward for a given episode is capped at 500 (i.e., 500 steps), after which the episode is truncated.

Figure 5.4 presents the training phase results for the three techniques, showing the average reward per episode over 10 repetitions within a total budget of 450 episodes. The results demonstrate that both RL agents (DQN and CRL) can converge to a suboptimal policy within the given budget: compared to the RANDOM baseline, both agents show an increasing

Table 5.2. RBST instantiation for CRL.

Step	**Choices**
Model building	Domain knowledge
Intervention variable selection	Fixed (action)
Intervention value assignment	Random
Effect estimation	Simulation-based

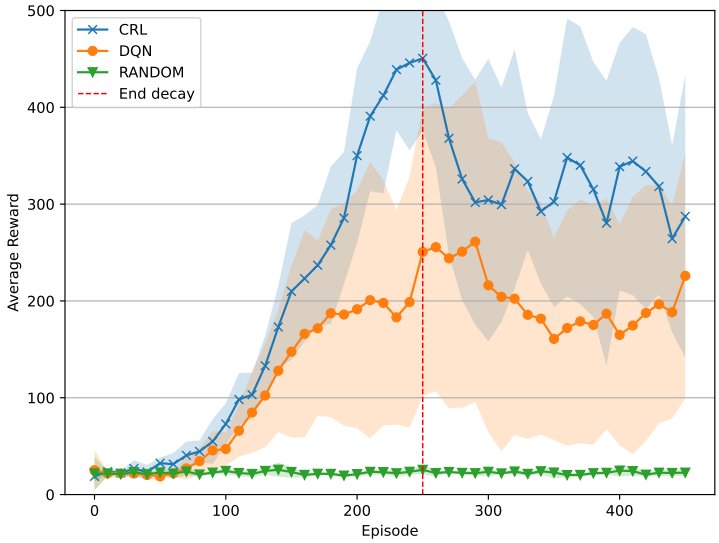


Figure 5.4. Average reward cartpole.

trend in average rewards over time. Moreover, CRL outperforms DQN in terms of efficiency, namely the average reward over time (episodes). CRL reaches the maximum average reward value of DQN in almost half budget and almost achieves a stable maximum average reward of 500 by the end of the exploration phase (i.e., ϵ decay completion). Although both RL agents show a slight performance drop during the exploitation phase, CRL stabilizes at a higher average reward (around 300) compared to DQN (around 200).

The results of this motivation study show that CRL can be a valid solution to improve the efficiency of the training phase of RL agents. It indeed boosts the exploration phase performance with the computation of counterfactual transitions. These findings can be very valuable in more complex environments like ADS testing where the state-action space can be impractical to efficiently explore.

5.3 Stateful testing of autonomous driving systems

In the literature, a common way to test ADSs is to manipulate the simulation environment where they operates [175]. As explained in Section 4.1 and Section 4.2, usually the objective of such techniques is to perturb the environment to try to cause misbehavior of the ADS, e.g., a collision with another vehicle or a traffic rule violation. Tests are generated automatically by solving an optimization problem, which consists of finding the optimal *static* configurations of the objects in the environment that challenge the ADS under test and optimize an objective function (usually the distance of the ADS from misbehavior). However, these techniques struggle to deal at runtime with sequential interactions, required to manipulate dynamic objects in the environment (e.g., another vehicle). On the other hand, the RL paradigm requires the agent to dynamically interact with the environment, learning from the effects of its actions. This offers an alternative way to test ADSs, by formulating the testing problem as an RL problem. The testing technique needs to choose a suitable RL algorithm to learn the actions that maximize the reward.

In this context, the use of RL for online testing of ADSs is gaining increasing interest. For instance, Haq *et al.* [56] proposed Many-Objective Reinforcement Learning for Online Testing (MORLOT), an online testing technique that combines RL and many-objective search to test the ADS module of an autonomous vehicle. MORLOT was evaluated in the CARLA simulation environment [36], a widely used high-fidelity driving simulator [175]. The ADS under test was the TransFuser model [146], the highest ranked ADS in the CARLA leaderboard [93] at the time of that study. The evaluation shows that MORLOT outperforms random testing as well as state-of-the-art search-based techniques, in terms of safety requirements violations exposed in a given time budget. However, in a own study [48] where these experiments were replicated, results pointed out no significant difference between MORLOT and the RANDOM baseline. Their extension study showed that further research is needed to fully address testing ADS with the use of RL. Besides Haq *et al.* [56], a relevant work is the one by Lu *et al.* [108]. Similarly to MORLOT, they propose a learning technique (DeepCollision) that dynamically changes the environ-

mental conditions to find collisions with vehicles, pedestrians, and static obstacles (corresponding to violations of requirements *r2*, *r3*, and *r4* in Section 4.3). DeepCollision uses DQN as RL agent to select actions from a set of 52 options to control the weather, time of the day, and behavior of actors (e.g., pedestrian crossing the road and vehicle switching lane). The state is defined by a set of 12 variables including traffic lights color, EV kinematics (position, speed, and rotation), and weather conditions. The authors employ the Apollo ADS [3] and the LGSVL simulator [155]. Unfortunately, LGSVL has been unmaintained since 2022, and the cloud servers are no longer operational.

Other techniques use Adaptive Stress Testing (AST), a method initially employed by Lee *et al.* [95] to test an aircraft collision avoidance system. AST formulates the problem of finding the most likely failure scenarios as a Markov decision process, which can be solved by RL agents. Koren *et al.* [88] explore the application of AST to find collisions in pedestrian crossing scenarios by extending it with deep RL. Corso *et al.* [27] also use AST, focusing on the reward formulation to find diverse and avoidable scenarios[2] as failing scenarios.

Sharif and Marijan [160] define a multi-agent environment in which a set of deep RL agents are trained with adversarial inputs. The goal is to find ADS failure states in which the EV goes off road or collides with obstacles. ADS robustness is then improved by retraining it with adversarial inputs.

The advances in applying reinforcement learning for online testing of ADSs are undeniable. However, the proposed techniques involve highly intricate and varied simulators, defining complex RL environments that require thorough study for drawing accurate conclusions.

5.4 Problem definition and notation

Similarly to the stateless formulation (Chapter 4) the ADS is embedded within the CARLA simulator [36], which renders a realistic town environment including junctions, vehicles, pedestrians, traffic lights, and traffic signs. The ADS that controls the EV has to drive it through a predefined

[2]They consider some scenarios to be unavoidable (e.g., a pedestrian causing a collision with a stopped ADS).

route (i.e., a scenario). However, in a stateful formulation, the scenario is not pre-defined at the start with all the settings kept static during execution. Instead, to allow a mapping to an RL problem, the scenario execution is divided into time steps. At each time step, the ADS receives and processes data from sensors (e.g., camera and LIDAR) to generate driving commands (steering, throttle, and braking) to maximize the driving performance. The route used for the evaluation consists of a 4-way intersection, governed by traffic lights, where multiple vehicles and pedestrians, which are driven by the embedded CARLA's autopilot[3], can approach from each direction.

The requirements definitions are the same as defined in Section 4.3. However, in this study, the focus is on a single objective problem, namely to find violations of *(r2)* (i.e., the EV must not collide with the other vehicles). The formulation of the testing as an RL problem follows the findings of own work [48]. Specifically, the state space has 23 variables consisting of the EV heading (radians), its speed on axes x and y, and the heading, relative distance and speed on axes x and y of the four closest vehicles to the EV. Relative ego-centric state variables can increase the generalization capabilities of the RL agent in those similar situations that require the same behavior [48, 98]. The action space of the RL agent is discrete: the agent can choose in a set of six meta-actions for each of the three vehicles closest to the EV, which change the behavior of their autopilots. Additionally, the agent has the option of "doing nothing", for a total of 19 actions. For each autopilot vehicle, the action set consists of: *i)* increment or decrement the target speed of the vehicle; *ii)* change lane, to the right/left; and *iii)* changing the offset from the center of the lane again, to the right/left. Furthermore, an Observation Time Period (OTP) [108] (also called *frame skipping* in the RL literature [11, 121]) is used. It consists of applying an action and waiting for it to produce an effect on the environment, by *pausing* the RL agent for a number of simulation steps (in the current setting, the RL agent acts at each simulator step). This way, the reward computed for a certain action comprises multiple simulation steps, giving more precise feedback to the agent. The OTP is set to

[3]These are not the same as the ADS under test, as their behavior is hardcoded depending on the state of the simulator. Differently from ADS, they do not rely solely on sensors.

skip 15 simulator ticks for every agent action. Worth stressing that the selected meta-actions are meant to provide a more effective convergence of the agent [98]. They create a layer of abstraction between the decision-making policy of the agent, and the low-level actions needed to control dynamic actors in the environment (i.e., low-level controls of throttle and steering angle for the vehicle) [97]. The low-level controller, running at a higher frame rate than the RL policy, takes care of translating the meta-actions to the actual commands, leaving the RL agent the responsibility to make only the most important and relevant decisions.

Finally, similarly to the one defined by Lu *et al.* [108], the reward function is based on the collision probability between EV and another vehicle (V). Specifically, the reward is computed on both the longitudinal and lateral safety distances. The *Longitudinal Safety Distance* ($LoSD$) measures the distance required to avoid a rear-end collision with the vehicle ahead V. It is computed using the following formula:

$$LoSD(v_{\text{EV}}, v_{\text{V}}) = \frac{1}{2}\left(\frac{v_{\text{EV}}^2}{a_{\text{EV}}} - \frac{v_{\text{V}}^2}{a_{\text{V}}}\right) + v_{\text{EV}}t + R_{\min}$$

Where v_{EV} is the speed of the EV, v_{V} is the speed of the leading vehicle V, respectively, with default values $a_{\text{EV}} = a_{\text{V}} = -6\,m/s^2$, t is the reaction time of the EV (set to 0 for autonomous driving), $R_{\min}$ is the minimum safe distance, set to 5 meters.

The *Lateral Safety Distance* ($LaSD$) measures the minimum lateral distance to avoid a side collision and is calculated as:

$$LaSD(v_{\text{EV}}) = \frac{(v_{\text{EV}} \cdot \sin\beta)^2}{a_{\text{EV}} \cdot \sin\beta}$$

Where v_{EV} is the velocity of the EV, β is the angle between the EV's direction and the other vehicle's position on the road, a_{EV} is the deceleration of the EV (set to $-6\,m/s^2$).

Using the computed safety distances, the *Collision Probability* ($ProC$) is determined by comparing the current distance (CD) between the EV and obstacles with the corresponding safety distance (SD):

$$ProC = \begin{cases} \frac{SD-CD}{SD} & \text{if } CD < SD \\ 0.0 & \text{otherwise} \end{cases}$$

Where SD can be either $LoSD$ or $LaSD$ depending on the situation (i.e., whether the vehicle V is in the same lane of the EV (LoSD) or not (LaSD), CD is the current distance between the EV and V, computed using the Euclidean distance:

$$CD = \sqrt{(x_{\text{EV}} - x_{\text{V}})^2 + (y_{\text{EV}} - y_{\text{V}})^2 + (z_{\text{EV}} - z_{\text{V}})^2}$$

The reward R_t at each time step is calculated based on the collision probability $ProC$ and designed to guide the RL agent towards actions that increase the likelihood of collisions. Specifically:

$$R_t = \begin{cases} -1.0 & \text{if } ProC < 0.2 \\ ProC & \text{if } 0.2 \leq ProC < 1.0 \\ 1.0 & \text{if a collision occurred} \end{cases}$$

Where a reward of -1.0 penalizes low collision probability, a reward of $ProC$ incentivizes increasing collision probability, a reward of 1.0 is given when a collision occurs.

The ADS under test is TransFuser++ [78], an evolution of Transfuser[146]. This ADS recently received one of the highest scores on the CARLA leaderboard and is expected to be much more reliable compared to its previous version. The TransFuser++ environment was configured following the instructions by the authors. The local configuration of the ADS was validated by running the agent in the intersection scenario and checking that no violation occurred (i.e., the ADS drives well in nominal conditions).

5.5 Preliminary Results

5.5.1 Evaluation criteria

As in the motivation study, the compared techniques are CRL, DQN, and RANDOM. The evaluation is based on *effectiveness* and *efficiency*. Effectiveness is computed as the total number of violations of the *r2*/DV requirement (collisions with other vehicles) after the entire testing budget, efficiency is computed as the average number of violations over time. Similarly to previous works by Haq *et al.*, the testing budget is set to 4 hours (240 minutes), with an ϵ-greedy policy with an exploration budget

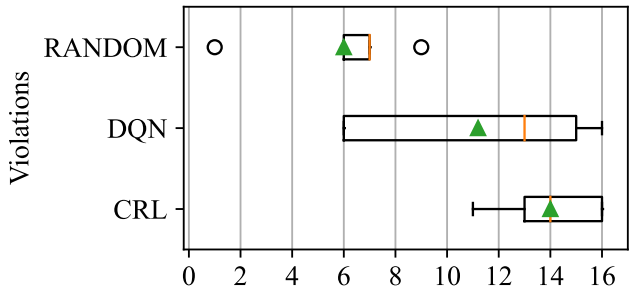


Figure 5.5. Effectiveness.

of 48 minutes. For each technique, 5 repetitions are run on the same 4-way intersection route, for a total of 60 computing hours.

5.5.2 Effectiveness

Figure 5.5 shows the box plots of the number of violations found by the three techniques after the entire 4 hours testing budget. It can be noticed that both RL agents, DQN and CRL, can converge to suboptimal policies, as they exhibit higher performance compared to the RANDOM baseline. Although DQN and CRL do not significantly differ in terms of the total number of violations, the second achieves a slightly higher median (14) compared to the first (13). Additionally, CRL shows a higher mean of 14 compared to the 11.2 of DQN, and with a lower standard deviation (respectively, ± 1.89 and ± 4.35).

5.5.3 Efficiency

Figure 5.5 shows the average number of violations found by the three techniques over time in the 5 repetitions. Similarly to the effectiveness investigation, both CRL and DQN achieve higher performance compared to the RANDOM baseline. This confirms the usefulness of RL in this problem formulation, as they achieve the same number of violations of RANDOM, on average, with half the budget. By comparing CRL and DQN, results confirm the superiority of the first, although less markedly than the difference found in the motivation study. CRL can achieve the

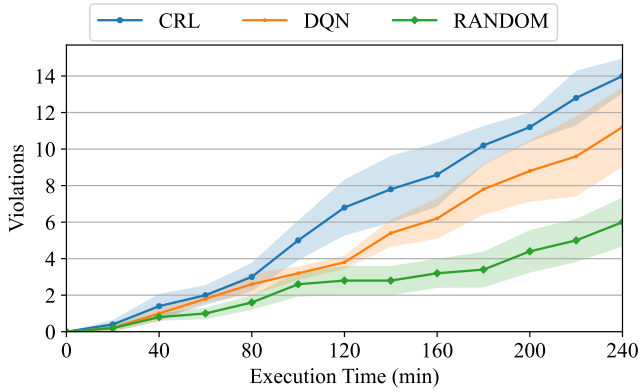


Figure 5.6. Efficiency.

Table 5.3. Area under the curve: Normalized Mean and Std.

Technique	Mean (±Std)
CRL	0.40 (±0.13)
DQN	0.29 (±0.10)
RANDOM	0.16 (±0.09)

same average number of violations of DQN in 40 minutes of budget less. Furthermore, at half-budget (i.e., 120 minutes) CRL obtained almost two times the number of violations compared to DQN.

To quantitatively evaluate the efficiency, the AUC of the average number of violations over time is measured. Table 5.3 reports the (min-max) normalized AUC mean and standard deviation per technique, in all routes. CRL covers 40% of the area, compared to 29% of DQN and 16% of RANDOM.

To further investigate the convergence capabilities of the RL agents, the distribution of the actions selected by each technique across the 5 repetitions has been extracted. In particular, in Figure 5.7 can be observed that, as expected, the actions selected by RANDOM follow a uniform distribution. On the other hand, CRL and DQN tend to privilege certain sets of actions given by the learned policy, despite the distributions including the exploration phase, when both sample actions uniformly at random. The

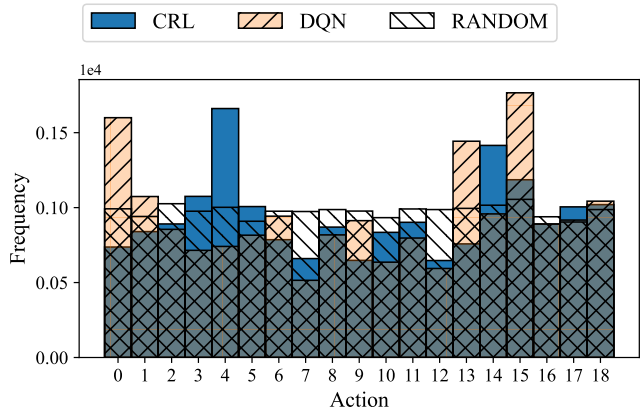


Figure 5.7. Action distribution.

two techniques mostly have the same distribution, by controlling the closest and the farthest vehicles (respectively, actions sets 0—5 and 12—17). However, they also focus on different actions, in these sets. Specifically, while DQN mainly focuses on accelerating/decelerating (actions 0—1 and 12—13), CRL prefers to control the lane changes and the lane offsets (actions 2—5 and 14—17).

5.5.4 Discussion

The instantiation of RBST for CRL in this study has demonstrated both advantages and limitations. One notable advantage is the method's enhanced efficiency during the exploration phase, where the CRL approach leverages counterfactual computations to improve and extend the RL agent body of knowledge. The ability to use hypothetical experience in CRL enables a focused search for critical states, although it can introduce a degree of bias due to reliance on simulated experiences rather than real-world data. This factor can be critical in notably unstable RL agents like DQN. For this reason, other solutions to improve stability and/or instantiate RBST with other RL agents are worth to be explored.

In general, while the proposed approach achieved higher performance in effectiveness and efficiency, particularly in simpler scenarios like the cartpole motivation study, it may benefit from further refinement in com-

plex, high-dimensional environments such as ADS, where the improvement is less marked and the study still at a preliminary stage. In this context, more time budget, additional RL agents, driving routes, and RBST instances would give a more comprehensive view of the benefits and limitations. Furthermore, currently, the causal model used assumes that each state variable has a potential effect on all others. This design choice, while conservative, introduces a level of generality that may not optimally suit all contexts. Future investigations could explore more domain-specific and tailored causal models or CD algorithms. Such refinements would reduce unnecessary connections between variables and align more closely with realistic dynamics. Moreover, the model-based planning approach applied, specifically through the Dyna architecture, was selected for its simplicity. It involves the generation of a single transition for each starting state, though longer *rollouts* - or even entire trajectories - can yield higher performance. While this choice facilitated the study's initial exploration, alternative model-based RL strategies could provide enhanced robustness and performance.

In summary, preliminary results suggest that RBST for CRL shows significant potential in enhancing stateful testing of ADS. However, optimizing causal models and exploring more advanced model-based planning methods are promising directions for future work.

5.6 Threats to validity

One significant concern is the dependency on simulated environments for experimentation: they may not fully reflect the complexity and variability of real-world ADS scenarios. This potential gap between the experimental environment and practical application could impact the generalizability of the findings.

The performance of the stateful RBST methodology is also closely tied to the reinforcement learning framework employed. The sensitivity of the model to hyperparameter tuning, exploration strategies, and learning rates introduces an inherent variability to the results. Differences in the reinforcement learning setup could lead to discrepancies in observed outcomes.

Moreover, the metrics used to evaluate the effectiveness and efficiency of the proposed approach are inherently limited in scope. While metrics

such as average reward, violation detection, and efficiency provide valuable insights, they may not capture the full spectrum of the system's behavior or its robustness under varying conditions.

Statistical limitations also pose a risk to the reliability of the findings. Limited test repetitions, variability in the test case generation, and the potential for noise in the results could undermine the robustness of the experimental outcomes. This could lead to challenges in confidently asserting the superiority or specific advantages of the RBST approach.

Despite these potential threats, efforts have been made to design the experiments rigorously, including the adoption of standardized evaluation criteria and detailed reporting of configurations and outcomes. However, further studies involving a broader range of software systems and varied testing environments are necessary to validate the approach and ensure its applicability to diverse contexts.

Chapter 6

Conclusions

The landscape of software testing has undergone a significant evolution, driven by the rapid growth of software complexity and autonomy.

Traditional testing approaches, based primarily on human intuition and exhaustive test case design, increasingly struggle to cope with the vast and dynamic input spaces of complex software systems. Scalability through automation has been enabled by data-driven testing methods, where ML plays a pivotal role. By reformulating testing as predictive tasks, these methods learn failure patterns and system behaviors from historical data, enabling a more efficient and cost-effective testing process. However, they are inherently limited: correlations learned on observations in a certain context are exploitable to make "predictions" solely based on what seen. Data-driven methods are limited to the first rung of the ladder of causation and fail to answer the fundamental question "What input causes the system to fail?"

This thesis presents a comprehensive exploration of RBST, a novel methodology that reformulates software testing as a causal reasoning task. By leveraging causal reasoning principles, RBST advances beyond traditional data-driven testing paradigms, enabling predictive and hypothesis-driven approaches. The methodology has been instantiated and evaluated for both stateless and stateful testing scenarios, with autonomous driving systems serving as the primary case study. The results underscore RBST's potential to transform software testing practices by enhancing the detection of safety violations and improving the coverage of safety-critical

scenarios compared to state-of-the-art machine learning and search-based techniques. In the stateful testing domain, the integration of causal reasoning with reinforcement learning frameworks demonstrated increased efficiency and effectiveness.

The contributions of this work address a critical limitation in current software testing practices: the reliance on correlation rather than causation. By reorienting the testing paradigm towards causality, RBST bridges this conceptual gap, enabling testers to ask and answer more fundamental questions about cause-effect relationships underlying system behavior. Moreover, the thesis provides a rigorous methodological framework for applying causal reasoning techniques in diverse testing contexts, supported by theoretical grounding and empirical evidence. The adaptability of RBST, as shown through its application to stateless and stateful scenarios, lays the groundwork for broader applicability across various domains of software systems. These achievements highlight RBST's role as a foundational methodology for advancing software testing in an increasingly complex and autonomous technological landscape.

Several open questions and research opportunities remain for future exploration. For example, one important research area concerns the scalability of RBST. While the proposed methodology has shown promising results, its performance and practicality in highly complex, real-world systems with vast state spaces require further investigation. Similarly, the dependency on high-quality causal models raises questions about RBST's robustness in scenarios where causal discovery processes are difficult and noisy. Future research could explore hybrid approaches that integrate domain knowledge to mitigate such challenges. Another concern lies in the computational overhead associated with causal reasoning. Although RBST has shown efficiency gains in generating targeted test cases, the computational cost of building and maintaining causal models, particularly in dynamic environments, could limit its adoption. Investigating methods for incremental or adaptive causal modeling might address this limitation. Additionally, the experiments in this thesis are mainly based on simulated environments. Although these controlled and reproducible settings provide valuable insights, they do not fully capture the complexity of real-world deployments. Validating RBST in operational environments, where factors such as unexpected interactions, unobserved variables, and emergent

behaviors play a significant role, is a critical next step. Furthermore, while autonomous driving systems were selected as the primary case study due to their complexity and relevance, this focus limits the generalizability of the findings. Extending RBST to other domains with different operational characteristics and testing requirements will be crucial in establishing its broader applicability.

Nevertheless, the insights gained from this thesis open exciting opportunities for advancing software testing methodologies. RBST's potential to address long-standing challenges in testing complex systems positions it as a foundational approach for future innovations. For instance, integrating RBST with explainable AI could enhance its applicability in safety-critical systems by providing transparent rationales for test outcomes. Combining RBST with continuous testing frameworks and real-time monitoring systems offers a pathway to fully adaptive testing solutions that evolve alongside the systems they assess. The emergence of learning-enabled systems and autonomous systems highlights a fertile ground for RBST applications, as these domains require an understanding of causal dynamics to ensure safety and reliability. Future work could explore the integration of RBST with cutting-edge machine-learning techniques, including meta-learning and transfer learning.

In conclusion, this thesis lays the foundation for a causality-driven revolution in software testing. By reframing testing as a reasoning task and equipping testers with tools to navigate the complex causal dynamics of modern software systems, RBST holds the potential to significantly enhance the quality, safety, and robustness of software systems in an increasingly autonomous and interconnected world. Further research and development will be key to realizing this vision, addressing open questions, mitigating limitations, and seizing opportunities.

Bibliography

[1] Self-driving cars market size, share, growth, report 2022-2030. https://www.precedenceresearch.com/self-driving-cars-market., 2022. n. 1779, Precedence Research, Accessed: 2022-08-31.

[2] D. Amalfitano, V. Riccio, N. Amatucci, V. De Simone, and A. R. Fasolino. Combining automated gui exploration of android apps with capture and replay through machine learning. *Information and Software Technology*, 105:95–116, 2019.

[3] Baidu Apollo. `https://github.com/ApolloAuto/apollo`, 2017.

[4] S. Athey and G. Imbens. Recursive partitioning for heterogeneous causal effects. *Proceedings of the National Academy of Sciences*, 113(27):7353–7360, 2016.

[5] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[6] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *40th International Conference on Software Engineering (ICSE)*, pages 1016–1026. ACM, 2018.

[7] M. Bagherzadeh, N. Kahani, and L. Briand. Reinforcement learning for test case prioritization. *IEEE Transactions on Software Engineering*, 48(8):2836–2856, 2022.

[8] P. Barbrook-Johnson and A. S. Penn. *Fuzzy Cognitive Mapping*, pages 79–95. Springer International Publishing, Cham, 2022.

[9] L. Barnett, A. B. Barrett, and A. K. Seth. Granger causality and transfer entropy are equivalent for gaussian variables. *Phys. Rev. Lett.*, 103:238701, Dec 2009.

[10] A. Basiri, L. Hochstein, N. Jones, and H. Tucker. Automating chaos experiments in production. In *41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 31–40. IEEE/ACM, 2019.

[11] M. Bellemare, J. Veness, and M. Bowling. Investigating contingency awareness using atari 2600 games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26 (1), pages 864–871, Palo Alto, CA, USA, 2012. AAAI Press.

[12] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 63–74. IEEE, 2016.

[13] R. Ben Abdessalem, A. Panichella, S. Nejati, L. C. Briand, and T. Stifter. Testing autonomous cars for feature interaction failures using many-objective search. In *33rd International Conference on Automated Software Engineering*, pages 143–154. ACM, 2018.

[14] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering*, pages 85–103, 2007.

[15] A. Bertolino and E. Marchetti. *Guide to the Software Engineering Body of Knowledge SWEBOK - Software testing (chapt.5)*. IEEE Computer Society, 2004.

[16] P. Blöbaum, P. Götz, K. Budhathoki, A. A. Mastakouri, and D. Janzing. Dowhy-gcm: An extension of dowhy for causal inference in graphical causal models, 2022.

[17] B. Busjaeger and T. Xie. Learning for test prioritization: An industrial case study. In *24th SIGSOFT International Symposium on Foundations of Software Engineering*, page 975–980. ACM, 2016.

[18] M. Caliendo and S. Kopeinig. Some practical guidance for the implementation of propensity score matching. *Journal of Economic Surveys*, 22(1):31–72, 2008.

[19] A. Calò, P. Arcaini, S. Ali, F. Hauer, and F. Ishikawa. Generating avoidable collision scenarios for testing autonomous driving systems. In *IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 375–386. IEEE, 2020.

[20] J. Chen, Z. Wu, Z. Wang, H. You, L. Zhang, and M. Yan. Practical accuracy estimation for efficient deep neural network testing. *ACM Trans. Softw. Eng. Methodol.*, 29(4), oct 2020.

[21] Z. Chen, Y. Kang, F. Gao, L. Yang, J. Sun, Z. Xu, P. Zhao, B. Qiao, L. Li, X. Zhang, Q. Lin, and M. Lyu. Aiops innovations of incident management for cloud services. In *Cloud Intelligence Workshop, The 34th AAAI Conference on Artificial Intelligence*, February 2020.

[22] D. M. Chickering. Learning equivalence classes of bayesian-network structures. *J. Mach. Learn. Res.*, 2:445–498, mar 2002.

[23] A. G. Clark, M. Foster, B. Prifling, N. Walkinshaw, R. M. Hierons, V. Schmidt, and R. D. Turner. Testing causality in scientific modelling software. *ACM Transactions on Software Engineering and Methodology*, 33(1), 2023.

[24] A. G. Clark, M. Foster, N. Walkinshaw, and R. M. Hierons. Metamorphic testing with causal graphs. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 153–164, 2023.

[25] D. Colombo, M. H. Maathuis, M. Kalisch, and T. S. Richardson. Learning high-dimensional directed acyclic graphs with latent and selection variables. *The Annals of Statistics*, 40(1):294–321, 2012.

[26] J. D Correa, S. Lee, and E. Bareinboim. Counterfactual transportability: A formal approach. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pages 4370–4390. PMLR, 17–23 Jul 2022.

[27] A. Corso, P. Du, K. Driggs-Campbell, and M. J. Kochenderfer. Adaptive stress testing with reward augmentation for autonomous vehicle validation. In *Intelligent Transportation Systems Conference (ITSC)*, pages 163–168. IEEE, 2019.

[28] D. R. Cox. *Planning of Experiments*. Wiley, New York, 1958.

[29] Rajeev H. D. and Sadek W. Causal effects in nonexperimental studies: Reevaluating the evaluation of training programs. *Journal of the American Statistical Association*, 94(448):1053–1062, 1999.

[30] Y. Dang, Q. Lin, and P. Huang. Aiops: Real-world challenges and research innovations. In *41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 4–5. IEEE/ACM, 2019.

[31] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., USA, 2001.

[32] K. Deb. *Multi-objective Optimization*, pages 403–449. Springer US, Boston, MA, 2014.

[33] Zhihong Deng, Jing Jiang, Guodong Long, and Chengqi Zhang. Causal reinforcement learning: A survey, 2023.

[34] N. Diamantopoulos, J. Wong, D. I. Mattos, I. Gerostathopoulos, M. Wardrop, T. Mao, and C. McFarland. Engineering for a science-centric experimentation platform. In *2020 ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, page 191–200. ACM, 2020.

[35] E. Dijkstra. Notes on structured programming. `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF`, 1970.

[36] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. CARLA: An open urban driving simulator. In *1st Annual Conference on Robot Learning*, volume 78, pages 1–16. Proceedings of Machine Learning Research (PMLR), 2017.

[37] O. J. Dunn. Multiple comparisons using rank sums. *Technometrics*, 6(3):241–252, 1964.

[38] V. H. S. Durelli, R. S. Durelli, S. S. Borges, A. T. Endo, M. M. Eler, D. R. C. Dias, and M. P. Guimarães. Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability*, 68(3):1189–1212, 2019.

[39] D. Dworak, F. Ciepiela, J. Derbisz, I. Izzat, M. Komorkiewicz, and M. Wójcik. Performance of lidar object detection deep learning architectures based on artificially generated point cloud data from carla simulator. In *24th International Conference on Methods and Models in Automation and Robotics (MMAR)*, pages 600–605. IEEE, 2019.

[40] F. Eberhardt. Introduction to the foundations of causal discovery. *International Journal of Data Science and Analytics*, 3(2):81–91, 2017.

[41] F. Eberhardt and R. Scheines. Interventions and causal inference. *Philosophy of Science*, 74(5):981–995, 2007.

[42] M. Ermuth and M. Pradel. Monkey see, monkey do: effective generation of gui tests with inferred macro events. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 82–93, New York, NY, USA, 2016. ACM.

[43] R. Feldt, S. Poulding, D. Clark, and S. Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 223–233. IEEE, 2016.

[44] J. Fischbach, T. Springer, J. Frattini, H. Femmer, A. Vogelsang, and D. Mendez. Fine-grained causality extraction from natural language requirements using recursive neural tensor networks. In *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)*, pages 60–69, 2021.

[45] M. Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32(200):675–701, 1937.

[46] A. Gambi, M. Mueller, and G. Fraser. Automatically testing self-driving cars with search-based procedural content generation. In *28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, page 318–328. ACM, 2019.

[47] Maxime Gasse, Damien Grasset, Guillaume Gaudron, and Pierre-Yves Oudeyer. Causal reinforcement learning using observational and interventional data, 2021.

[48] L. Giamattei, M. Biagiola, R. Pietrantuono, S. Russo, and P. Tonella. Reinforcement learning for online testing of autonomous driving systems: a replication and extension study, 2024.

[49] C. Glymour, K. Zhang, and P. Spirtes. Review of causal discovery methods based on graphical models. *Frontiers in Genetics*, 10, 06 2019.

[50] I. Gog, S. Kalra, P. Schafhalter, M. A. Wright, J. E. Gonzalez, and I. Stoica. Pylot: A modular platform for exploring latency-accuracy tradeoffs in autonomous vehicles. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 8806–8813. IEEE, 2021.

[51] C. W. J. Granger. Investigating causal relations by econometric models and cross-spectral methods. *Econometrica*, 37(3):424–438, 1969.

[52] A. Guerriero, R. Pietrantuono, and S. Russo. Operation is the Hardest Teacher: Estimating DNN Accuracy Looking for Mispredictions. In *43rd International Conference on Software Engineering (ICSE)*, pages 348–358. IEEE, 2021.

[53] R. Guo, L. Cheng, J. Li, P. R. Hahn, and H. Liu. A Survey of Learning Causality with Data: Problems and Methods. *ACM Computing Surveys*, 53(4), 2020.

[54] O. Hamdi, A. Ouni, E. A. AlOmar, M. Ó Cinnéide, and M. W. Mkaouer. An empirical study on the impact of refactoring on quality metrics in Android applications. In *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, pages 28–39, 2021.

[55] F. U. Haq, D. Shin, and L. Briand. Efficient Online Testing for DNN-Enabled Systems using Surrogate-Assisted and Many-Objective Optimization. In *44th International Conference on Software Engineering (ICSE)*, pages 811–822. ACM, 2022.

[56] F. U. Haq, D. Shin, and L. Briand. Many-Objective Reinforcement Learning for Online Testing of DNN-Enabled Systems. In *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1814–1826. IEEE, 2023.

[57] F. U. Haq, D. Shin, S. Nejati, and L. Briand. Can offline testing of deep neural networks replace their online testing? a case study of automated driving systems. *Empirical Softw. Engg.*, 26(5), sep 2021.

[58] F. U. Haq, D. Shin, S. Nejati, and L. C. Briand. Comparing offline and online testing of deep neural networks: An autonomous car case study. In *IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 85–95. IEEE, 2020.

[59] M. Harman, Y. Jia, and Y. Zhang. Achievements, open problems and challenges for search based software testing. In *8th International Conference on Software Testing, Verification and Validation*, pages 1–12. IEEE, 2015.

[60] L. Harries, R. Storan Clarke, T. Chapman, S. V. P. L. N. Nallamalli, L. Ozgur, S. Jain, A. Leung, S. Lim, A. Dietrich, J. M. Hernández-Lobato, T. Ellis, C. Zhang, and K. Ciosek. Drift: Deep reinforcement learning for functional software testing, 2020.

[61] M. J. Harrold. Testing: a roadmap. In *Conference on The Future of Software Engineering*, ICSE '00, page 61–72, New York, NY, USA, 2000. ACM.

[62] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon. Comparing white-box and black-box test prioritization. In *38th International Conference on Software Engineering (ICSE)*, page 523–534. ACM, 2016.

[63] M. Á. Hernán, B. Brumback, and J. M. Robins. Marginal Structural Models to Estimate the Causal Effect of Zidovudine on the Survival of HIV-Positive Men. *Epidemiology*, 11(5), 2000.

[64] N. Hewage and D. Meedeniya. Machine learning operations: A survey on mlops tool support, 2022.

[65] C. Hitchcock. Causal Models. In E. N. Zalta and U. Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2024 edition, 2024.

[66] P. W. Holland. Statistics and causal inference. *Journal of the American Statistical Association*, 81(396):945–960, 1986.

[67] Y. Huang and M. Valtorta. Pearl's calculus of intervention is complete, 2012.

[68] IEEE-1044. Ieee standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, 2010.

[69] IEEE-610. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[70] IEEE-730. Ieee standard for software quality assurance processes. *IEEE Std 730-2014 (Revision of IEEE Std 730-2002)*, pages 1–138, 2014.

[71] G. W. Imbens. Nonparametric Estimation of Average Treatment Effects Under Exogeneity: A Review. *The Review of Economics and Statistics*, 86(1):4–29, 02 2004.

[72] G. W. Imbens. Potential Outcome and Directed Acyclic Graph Approaches to Causality: Relevance for Empirical Practice in Economics. *Journal of Economic Literature*, 58(4):1129–79, 2020.

[73] C. Ioannides and K. I. Eder. Coverage-directed test generation automated by machine learning – a review. *ACM Trans. Des. Autom. Electron. Syst.*, 17(1), January 2012.

[74] ISO-24765. ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary. *ISO/IEC/IEEE 24765:2017(E)*, pages 1–541, 2017.

[75] ISO-9000. *ISO 9000 : international standards for quality management*. International Organization for Standardization Genève, Switzerland, Genève, Switzerland, 2nd ed edition, 1992.

[76] ISO/IEC 25010. ISO/IEC 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models, 2011.

[77] A. Jaber, M. Kocaoglu, K. Shanmugam, and E. Bareinboim. Causal discovery from soft interventions with unknown targets: Characterization and learning. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9551–9561. Curran Associates, Inc., 2020.

[78] Bernhard Jaeger, Kashyap Chitta, and Andreas Geiger. Hidden biases of end-to-end driving models. In *Proc. of the IEEE International Conf. on Computer Vision (ICCV)*, 2023.

[79] Z. Ji, P. Ma, S. Wang, and Y. Li. Causality-aided trade-off analysis for machine learning fairness. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 371–383. IEEE, sep 2023.

[80] Z. Ji, P. Ma, Y. Yuan, and S. Wang. Cc: Causality-aware coverage criterion for deep neural networks. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1788–1800, 2023.

[81] Y. Jin. Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 1(2):61–70, 2011.

[82] E. Kiciman and A. Sharma. Causal Reasoning: Fundamentals and Machine Learning Applications. https://causalinference.gitlab.io/, 2019.

[83] J. Kim, R. Feldt, and S. Yoo. Guiding Deep Learning System Testing Using Surprise Adequacy. In *41st International Conference on Software Engineering*, ICSE, pages 1039–1049. IEEE, 2019.

[84] D. P. Kingma and M. Welling. Auto-encoding variational bayes, 2022.

[85] M. Klischat and M. Althoff. Generating critical test scenarios for automated vehicles with evolutionary algorithms. In *IEEE Intelligent Vehicles Symposium (IV)*, pages 2352–2358. IEEE, 2019.

[86] M. Kocaoglu, A. Jaber, K. Shanmugam, and E. Bareinboim. Characterization and learning of causal graphs with latent variables from soft interventions. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *32nd Conference on Neural Information Processing Systems (NeurIPS)*, pages 14346–14356. Curran Associates, Inc., 2019.

[87] Z. Kong and C. Liu. Generating adversarial fragments with adversarial networks for physical-world implementation. *CoRR*, abs/1907.04449, 2019.

[88] M. Koren, S. Alsaif, R. Lee, and M. J. Kochenderfer. Adaptive stress testing for autonomous vehicles. In *Intelligent Vehicles Symposium*, pages 1–7. IEEE, 2018.

[89] B. Kosko. Fuzzy cognitive maps. *International Journal of Man-Machine Studies*, 24(1):65–75, 1986.

[90] D. Kumor, J. Zhang, and E. Bareinboim. Sequential causal imitation learning with unobserved confounders, 2022.

[91] A. Kwiatkowski, M. Towers, J. Terry, J. U. Balis, G. De Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. Jet Tai, H. Tan, and O. G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024.

[92] R. J. LaLonde. Evaluating the econometric evaluations of training programs with experimental data. *The American Economic Review*, 76(4):604–620, 1986.

[93] CARLA Autonomous Driving Leaderboard. CARLA leaderboard. `https://leaderboard.carla.org/leaderboard/`, 2020. Accessed: February 5, 2025.

[94] M. Lechner. The estimation of causal effects by difference-in-difference methods. *Foundations and Trends® in Econometrics*, 4(3):165–224, 2011.

[95] R. Lee, M. J. Kochenderfer, O. J. Mengshoel, G. P. Brat, and M. P. Owen. Adaptive stress testing of airborne collision avoidance systems. In *34th Digital Avionics Systems Conference*, pages 6C2–1–6C2–13. IEEE/AIAA, 2015.

[96] S. Lee and E. Bareinboim. Characterizing optimal mixed policies: Where to intervene and what to observe. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 8565–8576. Curran Associates, Inc., 2020.

[97] E. Leurent. An environment for autonomous driving decision-making. `https://github.com/eleurent/highway-env`, 2018.

[98] E. Leurent. A survey of state-action representations for autonomous driving. 2018.

[99] G. Li, Y. Li, S. Jha, T. Tsai, M. Sullivan, S. K. S. Hari, Z. Kalbarczyk, and R. Iyer. Av-fuzzer: Finding safety violations in autonomous driving systems. In *IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 25–36. IEEE, 2020.

[100] J. Li, X. Cui, Y. Wang, and F. Xie. An empirical study of software testing quality based on natural experiments. In *IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion*, pages 499–508, 2022.

[101] Y. Li, Z. M. J. Jiang, H. Li, A. E. Hassan, C. He, R. Huang, Z. Zeng, M. Wang, and P. Chen. Predicting node failures in an ultra-large-scale cloud computing platform: An aiops solution. *ACM Trans. Softw. Eng. Methodol.*, 29(2), April 2020.

[102] Z. Li, X. Ma, C. Xu, C. Cao, J. Xu, and J. Lü. Boosting Operational DNN Testing Efficiency through Conditioning. In *Proc. 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 499–509. ACM, 2019.

[103] P. Liu, Y. Li, B. Swain, and J. Huang. PUS: A Fast and Highly Efficient Solver for Inclusion-based Pointer Analysis. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1781–1792, 2022.

[104] Q. Liu, X. Wu, Q. Lin, J. Ji, and K. Wong. A novel surrogate-assisted evolutionary algorithm with an uncertainty grouping based infill criterion. *Swarm and Evolutionary Computation*, 60:100787, 2021.

[105] Y. Liu, D. I. Mattos, J. Bosch, H. H. Olsson, and J. Lantz. Bayesian propensity score matching in automotive embedded software engineering. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, pages 233–242, 2021.

[106] C. Louizos, U. Shalit, J. Mooij, D. Sontag, R. Zemel, and M. Welling. Causal effect inference with deep latent-variable models. In *31st International Conference on Neural Information Processing Systems*, page 6449–6459, Red Hook, NY, USA, 2017. Curran Associates Inc.

[107] C. Lu, B. Schölkopf, and J. M. Hernández-Lobato. Deconfounding reinforcement learning in observational settings, 2018.

[108] C. Lu, Y. Shi, H. Zhang, M. Zhang, T. Wang, T. Yue, and S. Ali. Learning configurations of operating environment of autonomous vehicles to maximize their collisions. *IEEE Transactions on Software Engineering*, 49(1):384–402, 2023.

[109] Y. Luo, X. Zhang, P. Arcaini, Z. Jin, H. Zhao, F. Ishikawa, R. Wu, and T. Xie. Targeting requirements violations of autonomous driving systems by dynamic evolutionary search. In *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 279–291. IEEE/ACM, 2021.

[110] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang. DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems. In *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, page 120–131. ACM, 2018.

[111] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang. DeepMutation: Mutation Testing of Deep Learning Systems. In *29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–111. IEEE, 2018.

[112] L. Ma, F. Zhang, M. Xue, B. Li, Y. Liu, J. Zhao, and Y. Wang. Combinatorial testing for deep learning systems. *CoRR*, abs/1806.07723, 2018.

[113] R. Maier, L. Grabinger, D. Urlhart, and J. Mottok. Causal models to support scenario-based testing of adas. *IEEE Transactions on Intelligent Transportation Systems*, 25(2):1815–1831, 2024.

[114] R. Majumdar, A. S. Mathur, M. Pirron, L. Stegner, and D. Zufferey. Paracosm: A language and tool for testing autonomous driving systems. *CoRR*, abs/1902.01084, 2019.

[115] D. Malinsky and D. Danks. Causal discovery algorithms: A practical guide. *Philosophy Compass*, 13(1):e12470, 2018.

[116] S. Mani and G. Cooper. Causal discovery from medical textual data. *Proceedings AMIA Symposium*, page 542, 12 2001.

[117] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In *5th International Conference on Software Testing, Verification and Validation*, pages 81–90, 2012.

[118] P. McMinn. Search-based software testing: Past, present and future. In *4th International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[119] K. Meinke, F. Niu, and M. Sindhu. Learning-based software testing: A tutorial. In Reiner Hähnle, Jens Knoop, Tiziana Margaria, Dietmar Schreiner, and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 200–219, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[120] B. Miranda, R. Verdecchia, E. Cruciani, and A. Bertolino. FAST approaches to scalable similarity-based test case prioritization. In *40th International Conference on Software Engineering (ICSE)*, page 222–232. ACM, 2018.

[121] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[122] T. M. Moerland, J. Broekens, A. Plaat, and C. M. Jonker. Model-based reinforcement learning: A survey. *Foundations and Trends® in Machine Learning*, 16(1):1–118, 2023.

[123] S. A. Mondal, P. Rv, S. Rao, and A. Menon. LADDERS: Log Based Anomaly Detection and Diagnosis for Enterprise Systems. *Annals of Data Science*, pages 1–19. , 2023.

[124] J. M. Mooij, S. Magliacane, and T. Claassen. Joint causal inference from multiple contexts. *Journal of Machine Learning Research*, 21(99):1–108, 2020.

[125] R. Moraffah, P. Sheth, M. Karami, A. Bhattacharya, Q. Wang, A. Tahir, A. Raglin, and H. Liu. Causal inference for time series analysis: problems, methods and evaluation. *Knowledge and Information Systems*, 63(12):3041–3085, Dec 2021.

[126] S. L. Morgan and C. Winship. *Counterfactuals and Causal Inference: Methods and Principles for Social Research*. Analytical Methods for Social Research. Cambridge University Press, 2 edition, 2014.

[127] G. E. Mullins, P. G. Stankiewicz, R. C. Hawthorne, and S. K. Gupta. Adaptive generation of challenging scenarios for testing and evaluation of autonomous vehicles. *Journal of Systems and Software*, 137:197–215, 2018.

[128] H. Namkoong, R. Keramati, S. Yadlowsky, and E. Brunskill. Off-policy policy evaluation for sequential decisions under unobserved confounding, 2020.

[129] J. Neyman. Sur les applications de la théorie des probabilités aux experiences agricoles: Essai des principes. *Roczn. Nauk Rolniczych*, 10:1–51, 1923.

[130] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, 1992.

[131] D.R. Niranjan, B. C. VinayKarthik, and Mohana. Deep learning based object detection model for autonomous driving research using carla simulator. In *2nd International Conference on Smart Electronics and Communication (ICOSEC)*, pages 1251–1258. IEEE, 2021.

[132] A. R. Nogueira, A. Pugnana, S. Ruggieri, D. Pedreschi, and J. Gama. Methods and tools for causal discovery and causal inference. *WIREs Data Mining and Knowledge Discovery*, 12(2):e1449, 2022.

[133] J. M. Ogarrio, P. Spirtes, and J. Ramsey. A Hybrid Causal Search Algorithm for Latent Variable Models. In A. Antonucci, G. Corani, and C. P. Campos, editors, *8th International Conference on Probabilistic Graphical Models*, volume 52 of *Proceedings of Machine Learning Research*, pages 368–379, Lugano, Switzerland, 06–09 Sep 2016. PMLR.

[134] S. Oh, S. Lee, and S. Yoo. Effectively sampling higher order mutants using causal effect. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 19–24, 2021.

[135] Y. Pang, X. Xue, and A. S. Namin. Identifying effective test cases through k-means clustering for enhancing regression testing. In *12th International Conference on Machine Learning and Applications*, volume 2, pages 78–83, 2013.

[136] A. Panichella, F. M. Kifetew, and P. Tonella. Reformulating branch coverage as a many-objective optimization problem. In *8th International Conference on Software Testing, Verification and Validation*, pages 1–10. IEEE, 2015.

[137] D. Parthasarathy and A. Johansson. Silgan: Generating driving maneuvers for scenario-based software-in-the-loop testing. In *IEEE International Conference on Artificial Intelligence Testing (AITest)*, pages 65–72. IEEE, 2021.

[138] K. Patel and R. M. Hierons. A mapping study on testing non-testable systems. *Software Quality Journal*, 26(4):1373–1413, dec 2018.

[139] J. Pearl. Probabilities of causation: Three counterfactual interpretations and their identification. *Synthese*, 121(1):93–149, Nov 1999.

[140] J. Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press, USA, 2nd edition, 2009.

[141] J. Pearl and D. Mackenzie. *The Book of Why: The New Science of Cause and Effect*. Basic Books, Inc., USA, 1st edition, 2018.

[142] K. Pei, Y. Cao, J. Yang, and S. Jana. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. *Commun. ACM*, 62(11):137–145, 2019.

[143] J. Peters, D. Janzing, and B. Schlkopf. *Elements of Causal Inference: Foundations and Learning Algorithms*. The MIT Press, 2017.

[144] M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles, and Techniques*. Wiley India Pvt. Limited, 2008.

[145] R. Pietrantuono, M. Ficco, and F. Palmieri. Testing the Resilience of MEC-Based IoT Applications Against Resource Exhaustion Attacks. *IEEE Transactions on Dependable and Secure Computing*, 21(2):804–818, 2024.

[146] A. Prakash, K. Chitta, and A. Geiger. Multi-modal fusion transformer for end-to-end autonomous driving. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7073–7083. IEEE, 2021.

[147] Alexandre Rafael Lenz, Aurora Pozo, and Silvia Regina Vergilio. Linking software testing results with a machine learning approach. *Engineering Applications of Artificial Intelligence*, 26(5):1631–1640, 2013.

[148] V. K. Raghu, J. D. Ramsey, A. Morris, D. V. Manatakis, P. Sprites, P. K. Chrysanthis, C. Glymour, and P. V. Benos. Comparison of strategies for scalable causal discovery of latent variable models from mixed data. *International Journal of Data Science and Analytics*, 6(1):33–45, Aug 2018.

[149] M. M. Rahman, I. Ceka, C. Mao, S. Chakraborty, B. Ray, and W. Le. Towards causal deep learning for vulnerability detection. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1–11. ACM, 2024.

[150] J. Ramsey, M. Glymour, R. Sanchez-Romero, and C. Glymour. A million variables and more: the Fast Greedy Equivalence Search algorithm for learning high-dimensional graphical causal models, with an application to functional magnetic resonance images. *International Journal of Data Science and Analytics*, 3, 2017.

[151] J. Ramsey, K. Zhan, M. Glymour, R. Sanchez Romero, B. Huang, I. Ebert-Uphoff, S. M. Samarasinghe, E. A. Barnes, and C. Glymour. TETRAD - A Toolbox for Causal Discovery. In *8th International Workshop on Climate Informatics*, 2018.

[152] V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, and P. Tonella. Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering*, 25(6):5193–5254, 2020.

[153] V. Riccio and P. Tonella. Model-Based Exploration of the Frontier of Behaviours for Deep Learning System Testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 876–888. ACM, 2020.

[154] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella. Deep reinforcement learning for black-box testing of android apps. *ACM Trans. Softw. Eng. Methodol.*, 31(4), July 2022.

[155] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta, E. Agafonov, T. H. Kim, E. Sterner, K. Ushiroda, M. Reyes, D. Zelenkovsky, and S. Kim. Lgsvl simulator: A high fidelity simulator for autonomous driving. In *23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–6. IEEE, 2020.

[156] P. R. Rosenbaum and D. B. Rubin. Reducing bias in observational studies using subclassification on the propensity score. *Journal of the American Statistical Association*, 79(387):516–524, 1984.

[157] D. B. Rubin. Estimating causal effects of treatments in randomized and nonrandomized studies. *Journal of Educational Psychology*, 66(5):688–701, 1974.

[158] D. B. Rubin. Randomization analysis of experimental data: The fisher randomization test comment. *Journal of the American Statistical Association*, 75(371):591–593, 1980.

[159] M. Salehie, L. Pasquale, I. Omoronyia, R. Ali, and B. Nuseibeh. Requirements-driven adaptive security: Protecting variable assets at runtime. In *2012 20th IEEE International Requirements Engineering Conference (RE)*, pages 111–120, 2012.

[160] A. Sharif and D. Marijan. Adversarial deep reinforcement learning for improving the robustness of multi-agent autonomous driving policies. In *29th Asia-Pacific Software Engineering Conference (APSEC)*, pages 61–70. IEEE, 2022.

[161] A. Sharma, C. Zhang, V. Syrgkanis, and E. Kiciman. Dowhy: Addressing challenges in expressing and validating causal assumptions. Technical Report MSR-TR-2021-15, ICML 2021 workshop on the Neglected Assumptions in Causal Inference, July 2021.

[162] Amit Sharma, Emre Kiciman, et al. DoWhy: A Python package for causal inference. https://github.com/microsoft/dowhy, 2019.

[163] S. Shimizu, P. O. Hoyer, A. Hyvärinen, and A. Kerminen. A linear non-gaussian acyclic model for causal discovery. *Journal of Machine Learning Research*, 7(72):2003–2030, 2006.

[164] D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanović, and M. Vokáč. *Challenges and Recommendations When Increasing the Realism of Controlled Software Engineering Experiments*, volume 2765 of *Lecture Notes in Computer Science*, pages 24–38. Springer, Berlin, Heidelberg, 2003.

[165] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 12–22, New York, NY, USA, 2017. ACM.

[166] P. Spirtes, C. Glymour, and R. Scheines. *Causation, Prediction, and Search.* Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, USA, 2nd edition, 2001.

[167] P. Spirtes and K. Zhang. Causal discovery and inference: concepts and recent methodological advances. *Applied Informatics*, 3(1):3, Feb 2016.

[168] J. Splawa-Neyman, D. M. Dabrowska, and T. P. Speed. On the application of probability theory to agricultural experiments. essay on principles. section 9. *Statistical Science*, 5(4):465–472, 1990.

[169] M. Steyvers, J. B. Tenenbaum, E. Wagenmakers, and B. Blum. Inferring causal networks from observations and interventions. *Cognitive Science*, 27(3):453–489, 2003.

[170] A. Stocco, B. Pulfer, and P. Tonella. Mind the gap! A study on the transferability of virtual vs physical-world testing of autonomous driving systems. *IEEE Transactions on Software Engineering (Early Access)*, pages 1–13, 2022.

[171] A. Stocco and P. Tonella. Confidence-driven weighted retraining for predicting safety-critical failures in autonomous driving systems. *Journal of Software: Evolution and Process*, 34(10):e2386, 2022.

[172] A. Stocco, M. Weiss, M. Calzana, and P. Tonella. Misbehaviour prediction for autonomous driving systems. In *42nd International Conference on Software Engineering*, page 359–371. ACM, 2020.

[173] R. S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *SIGART Bull.*, 2(4):160–163, jul 1991.

[174] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* A Bradford Book, Cambridge, MA, USA, 2018.

[175] S. Tang, Z. Zhang, Y. Zhang, J. Zhou, Y. Guo, S. Liu, S. Guo, Y. Xue, et al. A survey on automated driving system testing: Landscapes and trends. *ACM Transactions on Software Engineering and Methodology*, 32(5):1–62, 2023.

[176] H. Theil. *Economic Forecasts and Policy.* North-Holland Publishing Company, Amsterdam, 1961.

[177] D. L. Thistlethwaite and D. T. Campbell. Regression-discontinuity analysis: An alternative to the ex post facto experiment. *Journal of Educational Psychology*, 51:309–317, 1960.

[178] Y. Tian, K. Pei, S. Jana, and B. Ray. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *40th International Conference on Software Engineering (ICSE)*, page 303–314. ACM, 2018.

[179] J. Togelius, A. J. Champandard, P. L. Lanzi, M. Mateas, A. Paiva, M. Preuss, and K. O. Stanley. Procedural Content Generation: Goals, Challenges and Actionable Steps. In *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 61–75. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2013.

[180] P. Tonella, P. Avesani, and A. Susi. Using the case-based ranking methodology for test case prioritization. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 123–133, 2006.

[181] C. Tran and E. Zheleva. Learning triggers for heterogeneous treatment effects. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):5183–5190, Jul. 2019.

[182] I. Tsamardinos, L. E. Brown, and C. F. Aliferis. The max-min hill-climbing bayesian network structure learning algorithm. *Machine Learning*, 65(1):31–78, Oct 2006.

[183] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski. Simulation-based adversarial test generation for autonomous vehicles with machine learning components. *CoRR*, abs/1804.06760, 2018.

[184] W. M. van der Wal and R. B. Geskus. ipw: An r package for inverse probability weighting. *Journal of Statistical Software*, 43(13):1–23, 2011.

[185] S. Wager and S. Athey. Estimation and inference of heterogeneous treatment effects using random forests. *Journal of the American Statistical Association*, 113(523):1228–1242, 2018.

[186] H. Wang, Y. Jin, and J. Doherty. Committee-based active learning for surrogate-assisted particle swarm optimization of expensive problems. *IEEE Transactions on Cybernetics*, 47(9):2664–2677, 2017.

[187] L. Wang, S. Huang, S. Wang, J. Liao, T. Li, and L. Liu. A survey of causal discovery based on functional causal model. *Engineering Applications of Artificial Intelligence*, 133:108258, 2024.

[188] L. Wang, Z. Yang, and Z. Wang. Provably efficient causal reinforcement learning with confounded observational data. In *35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA, 2024. Curran Associates Inc.

[189] C. J. C. H. Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, UK, 1989.

[190] W. Webber, A. Moffat, and J. Zobel. A similarity measure for indefinite rankings. *ACM Trans. Inf. Syst.*, 28(4), nov 2010.

[191] E. J. Weyuker. Assessing test data adequacy through program inference. *ACM Trans. Program. Lang. Syst.*, 5(4):641–655, October 1983.

[192] M. Wicker, X. Huang, and M. Kwiatkowska. Feature-guided black-box safety testing of deep neural networks. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 408–426, Cham, 2018. Springer International Publishing.

[193] C. K. Wongchokprasitti, H. Hochheiser, J. Espino, E. Maguire, B. Andrews, M. Davis, and C. Inskip. bd2kccd/py-causal v1.2.1, December 2019.

[194] W. Wu, H. Xu, S. Zhong, M. R. Lyu, and I. King. Deep Validation: Toward Detecting Real-World Corner Cases for Deep Neural Networks. In *49th Annual IEEE/IFIP Int. Conference on Dependable Systems and Networks*, DSN, pages 125–137. IEEE, 2019.

[195] Y. Yang, X. Xia, D. Lo, and J. Grundy. A survey on deep learning for software engineering. *ACM Comput. Surv.*, 54(10s), September 2022.

[196] Z. Yang, Y. Chai, D. Anguelov, Y. Zhou, P. Sun, D. Erhan, S. Rafferty, and H. Kretzschmar. Surfelgan: Synthesizing realistic sensor data for autonomous driving. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11115–11124. IEEE, 2020.

[197] L. Yao, Z. Chu, S. Li, Y. Li, J. Gao, and A. Zhang. A survey on causal inference. *ACM Transactions on Knowledge Discovery from Data*, 15(5), 2021.

[198] Y. Yu, J. Chen, T. Gao, and M. Yu. DAG-GNN: DAG Structure Learning with Graph Neural Networks. In K. Chaudhuri and R. Salakhutdinov, editors, *36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7154–7163. PMLR, 2019.

[199] Y. Yu, X. Li, K. Bu, Y. Chen, and J. Yang. Falcon: Differential fault localization for sdn control plane. *Computer Networks*, 162:106851, 2019.

[200] Y. Zeng, R. Cai, F. Sun, L. Huang, and Z. Hao. A survey on causal reinforcement learning, 2023.

[201] J. Zhang and E. Bareinboim. Bounding causal effects on continuous outcome. *AAAI Conference on Artificial Intelligence*, 35(13):12207–12215, May 2021.

[202] J. Zhang and E. Bareinboim. Can humans be out of the loop? In Bernhard Schölkopf, Caroline Uhler, and Kun Zhang, editors, *Proceedings of the First Conference on Causal Learning and Reasoning*, volume 177 of *Proceedings of Machine Learning Research*, pages 1010–1025. PMLR, 11–13 Apr 2022.

[203] J. Zhang and E. Bareinboim. Online reinforcement learning for mixed policy scopes. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 3191–3202. Curran Associates, Inc., 2022.

[204] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid. DeepRoad: GAN-Based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 132–142. ACM, 2018.

[205] X. Zheng, B. Aragam, P. K. Ravikumar, and E. P. Xing. DAGs with NO TEARS: Continuous Optimization for Structure Learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31, pages 1–12. Curran Associates, Inc., 2018.

[206] Z. Zhong, Z. Hu, S. Guo, X. Zhang, Z. Zhong, and B. Ray. Detecting multi-sensor fusion errors in advanced driver-assistance systems. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, page 493–505. ACM, 2022.

[207] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu. DeepBillboard: Systematic Physical-World Testing of Autonomous Driving Systems. In *42nd International Conference on Software Engineering (ICSE)*, pages 347–358. ACM, 2020.

[208] Z. Zhou, Y. Soon Ong, P. B. Nair, A. J. Keane, and K. Y. Lum. Combining global and local surrogate models to accelerate evolutionary optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 37(1):66–76, 2007.

Author's publications

1. L. Giamattei, A. Guerriero, R. Pietrantuono and S. Russo. Assessing Black-box Test Case Generation Techniques for Microservices, Quality of Information and Communications Technology (QUATIC), 2022, Communications in Computer and Information Science, vol 1621, Springer, 10.1007/978-3-031-14179-9_4;

2. L. Giamattei, A. Guerriero, R. Pietrantuono and S. Russo, Automated Grey-Box Testing of Microservice Architectures, 22nd International Conference on Software Quality, Reliability and Security (QRS), 2022, pp. 640-650, IEEE, 10.1109/QRS57517.2022.00070;

3. L. Giamattei, R. Pietrantuono and S. Russo, Reasoning-Based Software Testing, 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), 2023, pp. 66-71, IEEE/ACM, 10.1109/ICSE-NIER58687.2023.00018;

4. L. Giamattei, A. Guerriero, R. Pietrantuono and S. Russo, Causality-driven Testing of Autonomous Driving Systems, ACM Transactions on Software Engineering and Methodology (TOSEM), 2024, 33, 3, Article 74, 35 pages, ACM, 10.1145/3635709;

5. M. Dinga, I. Malavolta, L. Giamattei, A. Guerriero, R. Pietrantuono, An Empirical Evaluation of the Energy and Performance Overhead of Monitoring Tools on Docker-Based Systems, International Conference on Service-Oriented Computing (ICSOC), 2023, Lecture Notes in Computer Science, vol 14419, Springer, 10.1007/978-3-031-48421-6_13;

6. L. Giamattei, A. Guerriero, R. Pietrantuono, S. Russo, I. Malavolta, T. Islam, M. Dînga, A. Koziolek, S. Singh, M. Armbruster, J.M. Gutierrez-Martinez, S. Caro-Alvaro, D. Rodriguez, S. Weber, J. Henss, E. Fernandez Vogelin, F. Simon Panojo, Monitoring tools for DevOps and microservices:

A systematic grey literature review, Journal of Systems and Software (JSS), 2023, Volume 208, 2024, 111906, ISSN 0164-1212, 10.1016/j.jss.2023.111906;

7. L. Giamattei, A. Guerriero, R. Pietrantuono and S. Russo, Automated functional and robustness testing of microservice architectures, Journal of Systems and Software (JSS), 2024, Volume 207, 111857, ISSN 0164-1212, 10.1016/j.jss.2023.111857;

8. L. Giamattei, A. Guerriero, I. Malavolta, C. Mascia, R. Pietrantuono, and S. Russo. Identifying Performance Issues in Microservice Architectures through Causal Reasoning. 5th International Conference on Automation of Software Test (AST), 2024, 149–153, IEEE/ACM, 10.1145/3644032.3644460;

9. M. S. Floroiu, S. Russo, L. Giamattei, A. Guerriero, I. Malavolta, R. Pietrantuono, Anomaly Detection and Root Cause Analysis of Microservices Energy Consumption, IEEE International Conference on Web Services (ICWS), 2024, pp. 590-600, 10.1109/ICWS62655.2024.00079.

10. L. Giamattei, M. Biagiola, R. Pietrantuono, S. Russo, P. Tonella, Reinforcement Learning for Online Testing of Autonomous Driving Systems: a Replication and Extension Study, Empirical Software Engineering (EMSE), 2025, 30, 19, 10.1007/s10664-024-10562-5

11. L. Giamattei, A. Guerriero, R. Pietrantuono, S. Russo, Causal reasoning in Software Quality Assurance: A systematic review, Information and Software Technology (IST), Volume 178, 2025, 107599, ISSN 0950-5849, 10.1016/j.infsof.2024.107599.