Establishing the Causal Foundations of Metamorphic Testing: A Novel Application of Causal Inference for Testing Computational Modelling Software

Andrew Graham Clark

The University of Sheffield

Faculty of Engineering

Department of Computer Science

Supervisor: Dr. Neil Walkinshaw

Second Supervisor: Prof. Robert M. Hierons

September 2023

A thesis submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

Abstract

From simulating galaxy formation to disease progression in a pandemic, computational models play an essential role in developing scientific theories and supporting government policy decisions that affect us all. Given these critical applications, a poor modelling assumption or bug could have far-reaching consequences. However, computational models suffer from a fundamental challenge that renders most testing techniques unsuitable - the test oracle problem: Given computational models predominantly simulate future events, how can we determine whether their outputs are correct?

Metamorphic testing is one technique that is widely used to mitigate the oracle problem. However, computational models also possess several properties that make this approach impractical at scale, including a complex input space, long execution times, and non-determinism. Consequently, there is a need to increase the applicability of metamorphic testing to computational modelling software.

To this end, in this thesis, we draw on the fundamentally causal nature of metamorphic testing and examine its overlap with causal inference - a family of statistical methodologies designed to establish salient cause-effect relationships without necessarily performing costly controlled experiments. We posit that metamorphic testing shares the same objective as causal inference and highlight a number of potential advantages of approaching the former as a problem of the latter. To investigate these potential advantages, we outline a conceptual software testing framework that introduces the fundamental components necessary to apply causal inference methods to causality-centred testing activities, such as metamorphic testing. We then provide an implementation of this framework and apply it to real-world computational modelling software, finding that causal inference methods can be used to accurately predict metamorphic test outcomes from small amounts of sparse and potentially biased data. In addition, using established graphical models of causality, we propose and evaluate a causal model-based approach for systematically generating metamorphic relations and tests capable of detecting bugs affecting causal structure.

Acknowledgements

First, I would like to thank my supervisors, Neil and Rob, for their invaluable guidance, patience, and support. Throughout my studies, Neil has gone above and beyond to support me in both an academic and personal capacity, always finding time to meet with me and discuss my concerns. It is thanks to these meetings that I could remain motivated and level-headed, even at the most difficult times. Rob has also been exceptionally generous with his time and regularly went out of his way to offer insights and advice that undoubtedly pushed me to become a better researcher. It has been a privilege to work with you both.

In addition, I would like to thank other members of the wider CITCoM research group and RSE team who have supported my research: Michael Foster, Nick Latimer, Richard Somers, Chris Wild, and Bob Turner. A special thanks to Michael who has been a great friend and mentor throughout my studies, and to Nick for his continual help with all things causal inference.

I would also like to thank my family for their unwavering support. In particular, I would like to thank: Alanna, for your endless patience and putting up with me for all this time; Mum, for always finding the time to talk and listen to my concerns; Dad, for our conversations about science that inspired me to pursue a Ph.D. in the first place; and Grandma, for always showing a genuine interest in our conversations about my research.

Finally, I would like to thank my friends from home, Josh and Rob, who made life outside of work so much more enjoyable during this time, and friends from the testing research group, especially Michael, Owain, and Richard.

Table of Contents

Glossary

ABM	Agent-Based Model
AI	Artificial Intelligence
ATE	Average Treatment Effect
CATE	Conditional Average Treatment Effect
CPDA	Causal Program Dependence Analysis
CPDM	Causal Program Dependence Model
CSG	Causal Software Graph
CTF	Causal Testing Framework
DAG	Directed Acyclic Graph
EBM	Equation-Based Model
FL	Fault Localisation
FOM	First Order Mutant
FSM	Finite State Machine
GPE	Gaussian Process Emulator
GPS	Generalised Propensity Score
GSA	Gated Single Assignment
HOM	Higher Order Mutant
ITE	Individual Treatment Effect
ODE	Ordinary Differential Equation

PS	Propensity Score
PUT	Program-Under-Test
RCT	Randomised Controlled Trial
RR	Risk Ratio
SHD	Structural Hamming Distance
SMT	Statistical Metamorphic Testing
SSHOM	Strongly Subsuming Higher Order Mutant
STF	Software Testing Framework
SUT	System-Under-Test

List of Figures

List of Tables

1 | Introduction

On the 23rd March 2020, the UK government implemented its first national lockdown in response to the rapidly unfolding COVID-19 (SARS-CoV-2) pandemic [6]. This unprecedented decision saw the introduction of numerous policies that aimed to contain the spread of the virus, including the closure of schools and all but essential workplaces. While these policies were successful in reducing the number of COVID-19 infections and fatalities, they also had a significant impact on livelihood and the economy. Prolonged school closures disrupted education for millions of children; mass social isolation resulted in a widespread worsening of mental health; and increased strain on the public health sector reduced access to critical health services which, in turn, led to missed diagnoses for diseases such as Alzheimer's [7]. Moreover, from an economic perspective, the COVID-19 pandemic came at an exceptional fiscal cost, with estimated public spending ranging from £310bn - £410bn [8], reversing the previous two years of GDP growth [9].

This momentous policy decision was driven to a large extent by computational models. Throughout the COVID-19 pandemic, computational models played an essential role in simulating various mitigation strategies and interventions in order to support policymakers. Perhaps most notably, at the beginning of the pandemic, a team of researchers led by Neil Ferguson developed CovidSim: an agent-based model designed to evaluate the impact of various COVID-19 mitigation strategies [10]. CovidSim is noteworthy as it was the subject of the infamous 'Report 9' [10] that was one of the main driving forces behind the initial lockdown [11]. In particular, this report suggested that even the most optimal mitigation strategies would result in hundreds of thousands of deaths and health services being overwhelmed. Therefore, based on the results of this simulation, it was recommended that the UK should employ a combination of stringent interventions capable of reversing rather than merely dampening the spread of COVID-19 [10]. Soon after these findings were published, the UK government announced the first UK lockdown.

With people losing their jobs overnight, there was a clear need for transparency and accountability: the general public needed confidence in the model to understand *why* such stringent restrictions were necessary. Consequently, Report 9 and the CovidSim model came under great scrutiny from both the general public and experts. This issue was exacerbated by the fact that, initially, the source code was unavailable to the public and the assumptions upon which these forecasts were made were not publicly accessible either:

> (Neil Ferguson; Mar 22, 2020[1])
> *I'm conscious that lots of people would like to see and run the pandemic simulation code we are using to model control measures against COVID-19. To explain the background - I wrote the code (thousands of lines of undocumented C) 13+ years ago to model flu pandemics...*

However, in the context of computational modelling, the question of confidence is a difficult one. When the goal is to simulate events that have not yet occurred, how can policymakers know whether the simulations are accurate or even informative? This is a seemingly impossible task, particularly during the early stages of a pandemic, where so little is known and a great deal of educated guess work is needed to compensate for the lack of data. Indeed, it was later found that the values for the reproductive number (R_0) of COVID-19 used in Report 9 were underestimates [12]. This is not to suggest that the initial model was flawed, but that the accuracy of models will undoubtedly improve as data becomes available.

Nonetheless, in order to build trust in a computational model, it remains essential to determine whether its behaviour is broadly ‘correct’, even (and especially) when parameter estimates are shrouded in uncertainty. One way that we can build trust in the model is through *software testing*, in which we employ various tools and techniques in order to reduce the risk associated with using software and improve its quality [4]. Given the critical applications of computational models such as CovidSim, the importance of testing in this context is self evident; a bug or faulty modelling assumption could have catastrophic consequences, including extreme fiscal costs and even loss of life.

However, when CovidSim was later refactored and made open source, the initial version was released with only a single “basic regression test”[2]. This test was designed to detect whether subsequent changes to the model cause the output of a specific simulation to change[3]. Other than this regression test, no other tests were available in CovidSim until unit tests and integration tests were added in later versions.

This lack of testing is not unique to CovidSim: scientific software is notoriously difficult to test and developers of this class of software often lack a formal background in software engineering [13]. Consequently, good software engineering practices that would help to identify bugs and poor modelling assumptions - such as testing, writing documentation, and making code open source - are seldom applied in the scientific context. To this end, during the COVID-19 pandemic, the British Computer Society made an open call to improve the standard of software development practices used to develop computational modelling software that informs critical policy decisions [14].

Arising from their exploratory nature, one of the fundamental barriers to testing computational models is the *test oracle problem* [15]. As alluded to previously, when pandemic models are used for the express purpose of simulating events which have not yet occurred, how can one determine whether the output is correct? In the field of software testing, this intrinsic challenge of determining whether, given some inputs, the software produces the *correct* output is known as the test oracle problem.

One promising solution to the test oracle problem is *metamorphic testing* [16]. The key idea behind this approach is that transforming the input space in a prescribed way should *cause* a change in the output space that can be anticipated. Therefore, where a test oracle is unattainable, we can instead seek to establish salient cause-effect relationships between the input space and output space of software. Such cause-effect relationships can be characterised succinctly as so-called *metamorphic relations*. For example, in CovidSim, we could define and test the following metamorphic relation: *increasing the reproductive number of COVID-19 (R_0) should cause the cumulative infections to increase.* We can test this relationship by executing CovidSim twice, once with a smaller R_0 and once with a higher R_0, and contrasting their respective outputs. This approach effectively alleviates the oracle problem as we

[1]Retrieved from https://twitter.com/neil_ferguson/status/1241835454707699713?lang=en

[2]As stated in the documentation of the initial refactored version of CovidSim: https://github.com/mrc-ide/covid-sim/blob/bd87d47/README.md#testing

[3]Regression test source code in the initial version: https://github.com/mrc-ide/covid-sim/blob/bd87d47/tests/regressiontest_UK_100th.py

can determine correctness of the implementation with respect to specified properties without needing to know the precise correct output for the individual executions.

While metamorphic testing is, in theory, a suitable solution to the test oracle problem, computational models have several challenging characteristics that hinder its applicability in practice. First, computational models often have *vast and complex input spaces*, comprising many input and output variables that share complex and often interconnected causal relationships. Second, computational models tend to have *long execution times*, making them expensive and time consuming to test. Third, and compounding this issue further, many computational models are *non-deterministic*, meaning multiple runs with the same input will yield different results. Collectively, these issues give rise to the following barriers that preclude the application of metamorphic testing to computational models:

- The complex nature of the input-output relationships makes it especially difficult to identify suitable metamorphic relations.

- The long execution times paired with the need to repeat tests multiple times mean that each metamorphic test is extremely costly and time consuming to run.

In fields such as epidemiology that, like metamorphic testing, are concerned with establishing salient cause-effect relationships, similar problems abound. For example, epidemiologists have long been concerned with the ill-effects of smoking, seeking answers to important causal questions such as *"Does smoking cause lung cancer?"* To answer such a question, ideally, an epidemiologist would conduct a randomised control trial which isolates the key cause-effect relationship by design. However, such an experiment could never be conducted for obvious ethical reasons (it would require forcing individuals to smoke).

From a causal perspective, metamorphic testing can be viewed similarly as a form of experiment that aims to isolate the cause-effect relationship of interest. Therefore, epidemiologists are ultimately faced with the same fundamental problems as metamorphic testing, albeit under a different guise:

- Epidemiologists want to establish the presence of complex causal relationships — Testers want to capture and test complex causal relationships as metamorphic relations.

- Due to ethical concerns, it is often infeasible for an epidemiologist to conduct an experiment capable of isolating the causal relationship of interest — Due to the practical issues surrounding computational models, metamorphic testing is often infeasible.

To address these problems, epidemiologists have turned to a family of statistical methodologies known as causal inference. These methodologies help domain experts to systematically design statistical procedures that can justifiably derive causal effects from observational data. Put simply, this makes it possible to emulate the outcome of the ideal randomised experiment where the actual experiment cannot be conducted.

Given that causal inference has enabled epidemiologists to overcome the same fundamental challenges that hinder the application of metamorphic testing to computational models, a question that naturally emerges is: *Can causal inference also be used to overcome these problems in the context of metamorphic testing?*

1.1 Aims and Objectives

The overarching aim of this thesis is to explore the relationship between causal inference and metamorphic testing, and to determine whether this relationship can be leveraged to improve the applicability of metamorphic testing to computational modelling software. Specifically, we aim to answer the following high-level research question:

Can causal inference help to improve the applicability of metamorphic testing to computational modelling software?

In order to answer this question, the main body of this thesis is divided into two halves.

In the first half, we focus on establishing a theoretical relationship between causal inference and metamorphic testing and explore how, conceptually, this formulation can benefit computational models. We capture this goal in the following high-level research objective:

RO1 To explore how, *in theory*, causal inference can increase the applicability of metamorphic testing to computational modelling software.

In order to achieve this objective, we first aim to explore the factors that currently hinder the application of metamorphic testing to computational modelling software and use this information to formulate a problem statement. With our problem statement in mind, we then propose a solution: a software testing framework that facilitates the application of causal inference methods to software testing activities, including metamorphic testing. We then aim to use this framework to highlight the potential advantages of formulating software testing problems, like metamorphic testing, as problems of causal inference. This leads to the following research questions:

RQ1.1 What are the main factors that hinder the application of metamorphic testing to computational modelling software?

RQ1.2 How can we introduce causal inference to a software testing framework?

RQ1.3 What are the potential advantages to formulating metamorphic testing as a problem of causal inference?

Then, in the second half of this thesis, we turn our attention from theory to practice. In particular, we aim to investigate how we can leverage this conceptual relationship between causal inference and metamorphic testing to improve the applicability of metamorphic testing to computational models in practice. We capture this second goal in the following high-level research objective:

RO2 To explore whether, *in practice*, causal inference can increase the applicability of metamorphic testing to computational modelling software.

In particular, we investigate whether causal inference can be leveraged to address two notable limitations of metamorphic testing that we identify in the first half of thesis: (1) the need to manually construct metamorphic relations, and (2) the potentially high cost associated with executing metamorphic tests. With this in mind, in order to achieve **RO2**, we seek answers to the following research questions:

RQ2.1 How can we use causal inference methods to help testers to construct metamorphic relations?

RQ2.2 How can we use causal inference methods to reduce the cost associated with executing metamorphic tests, particularly when dealing with costly non-deterministic systems?

1.2 Publications and Software

Throughout this thesis, the following works have been published in peer-reviewed conferences and journals:

P1 **Clark, Andrew G.**, Neil Walkinshaw, and Robert M. Hierons. "Test case generation for agent-based models: A systematic literature review." Information and Software Technology 135 (2021): 106567.

P2 **Clark, Andrew G.**, Michael Foster, Neil Walkinshaw, and Robert M. Hierons. "Metamorphic Testing with Causal Graphs." In 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 153-164. IEEE, 2023.

P3 **Clark, Andrew G.**, Michael Foster, Benedikt Prifling, Neil Walkinshaw, Robert M. Hierons, Volker Schmidt, and Robert D. Turner. "Testing Causality in Scientific Modelling Software." ACM Trans. Softw. Eng. Methodol. Just Accepted (July 2023). https://doi.org/10.1145/3607184

P4 Anness, A. R., **A. Clark**, K. Melhuish, F. M. T. Leone, M. W. Osman, D. Webb, T. Robinson, N. Walkinshaw, A. Khalil, and H. A. Mousa. "Maternal hemodynamics and neonatal birth weight in pregnancies complicated by gestational diabetes: new insights from novel causal inference analysis modeling." Ultrasound in Obstetrics & Gynecology 60, no. 2 (2022): 215-222.

P5 Somers, Richard J., **Andrew G. Clark**, Neil Walkinshaw, and Robert M. Hierons. "Reliable counterparts: efficiently testing causal relationships in digital twins." In Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 468-472. 2022.

In addition, over the course of this thesis, the following open source software and software contributions have been made:

S1 Causal Testing Framework.
GitHub Repository: https://github.com/CITCOM-project/CausalTestingFramework.
This framework is an implementation of the conceptual framework that we introduce in Chapter 4. **Note:** the physical ownership of this framework has been transferred to the CITCoM research group who have since developed several new causal inference-led software testing methods based on the framework I originally developed. This thesis will only discuss my own contributions to this framework.

S2 Microsoft DoWhy: Improved Confidence Intervals.
Pull Request: https://github.com/py-why/dowhy/pull/278
In the earlier stages of this thesis, we initially used a causal inference framework developed by Microsoft (DoWhy). While using this framework, a bug was found in the confidence intervals method. In an open source contribution, we fixed this bug and implemented a better method of bootstrapping confidence intervals (empirical bootstrapping instead of percentile) that can obtain tighter confidence intervals.

S3 CosmicRay: `VariableInserter` and `VariableRemover` Mutation Operators.
Pull Request: https://github.com/sixty-north/cosmic-ray/pull/529
To evaluate the metamorphic relation generation approach introduced in Chapter 5, we required a way of mutating programs with a known causal structure to add and remove specific cause-

effect relationships modelled as variable assignments. We achieved this by implementing a pair of mutation operators in the Python mutation testing framework CosmicRay. We realised this could be a potentially useful operator outside of our experimental setting and therefore decided to generalise our implementation, as far as possible, to enable users to manipulate programs by adding and deleting named variables.

1.3 Thesis Outline and Contributions

The remainder of this thesis is structured as follows.

In Chapter 2, we present a background into the key topics concerning this thesis, including software testing (Section 2.1), causal inference (Section 2.2), and the intersection between the two (Section 2.3).

We then proceed to Part I, in which we outline our problem statement and proposed solution.

In Chapter 3, we introduce two real-world epidemiological models as motivating examples (Sections 3.1 and 3.2) and, broadly following existing work, demonstrate the application of metamorphic testing to this software (Section 3.3). Here, we highlight a number of limitations of metamorphic testing when applied to computational modelling software such as this, ultimately leading us to outline our problem statement for the thesis in Section 3.4. In this chapter, we make the following contributions:

- We present an in-depth application of metamorphic testing to two real-world epidemiological models, in which we use metamorphic relations designed for epidemiological models published in previous work.
- In the process, we reveal several limitations of metamorphic testing in this context and weaknesses of the existing metamorphic relations, and identify a potential bug in a real-world epidemiological model that is one of our motivating examples.

With our problem statement in mind, in Chapter 4, we proceed to outline our proposed solution: a causal inference-driven software testing framework. To motivate this decision, in Section 4.1, we start by discussing the overlap between metamorphic testing and causal inference, and elaborate on how causal inference has been used to solve similar problems to those outlined in the previous chapter, but in the field of epidemiology. Then, in Section 4.2, we outline the necessary causal components that must be added to a conventional software testing framework to facilitate the application of causal inference. In this chapter, which is partially based on publication [**P3**], we make the following contributions:

- We extend the conceptual software testing framework proposed by Staats et al. [1] (see Section 2.1.1) to include the causal components necessary to apply causal inference methods to software testing problems, forming the Causal Testing Framework.
- We establish a relationship between causal inference and metamorphic testing, and show how the latter can be expressed and solved as a problem of the former.

Having now established a link between causal inference and metamorphic testing, and introduced a conceptual framework that, in theory, facilitates the application of causal inference methods to software testing problems, in Part II, we turn our attention to practical matters. Specifically, in this part of the thesis, we begin to exploit this relationship to understand how causal inference methods can be leveraged to improve metamorphic testing and, in particular, its applicability to computational modelling software.

To this end, in Chapter 5, we investigate how graphical causal models (discussed in Section 2.2.6) can be used as a form of causal specification that supports the generation of metamorphic relations. This chapter is motivated by one of the key barriers to applying metamorphic testing: the need to manually identify metamorphic relations and is based on publication [**P2**]. In this chapter, we make the following contributions:

- We introduce a novel causal model-based approach for the generation of metamorphic relations.
- Through a series of experiments, we demonstrate that the proposed approach is relatively robust to misspecifications in the underlying causal DAG and can detect "evasive" conditional causal relationships where an appropriate test generation strategy is employed.
- We demonstrate the applicability of this approach to real-world software by conducting two illustrative case studies involving bugs that affect causal structure from the Defects4J framework [17].

Following this, in Chapter 6, we focus on another key barrier to applying metamorphic testing that is especially problematic for computational modelling software: the cost associated with executing metamorphic tests. To address this, in this chapter, we provide a reference implementation of the causality-focused software testing framework introduced in Chapter 4 and demonstrate in a pair of real-world case studies how this can be used to conduct metamorphic testing as a problem of causal inference. In particular, we show that, where computational executions are prohibitively expensive, we can use potentially confounded observational data and causal inference methods to accurately predict metamorphic test outcomes. In this chapter, which is based on the case studies conducted in publication [**P3**], we make the following contributions:

- We provide a reference implementation of the Causal Testing Framework: a software testing framework that facilitates the application of observational and experimental causal inference methods.
- We conduct a pair of real-world case studies that demonstrate the accuracy, efficiency, and effort involved in applying the framework. In conducting these case studies, we also uncover a notable bug in a real-world epidemiological model that was later acknowledged and fixed by its developers.
- We demonstrate the feasibility of using causal inference methods to conduct causal testing activities, like metamorphic testing, using observational data *a posteriori*.

Finally, in Chapter 7, we conclude this thesis by reflecting on the research objectives and answering the high-level research questions outlined in Section 1.1. In addition, based on the key findings, we outline directions for future work into the intersection of causal inference and software testing.

2 | Background

In this chapter, we present a background into the main topics that concern this thesis, including software testing, causal inference, and the intersection of the two.

We start in Section 2.1 with a discussion of software testing, including the fundamental components of an archetypal software testing framework (Section 2.1.1), measures and methods of measuring test adequacy (Section 2.1.3), the test oracle problem (Section 2.1.4), and metamorphic testing (Section 2.1.5).

Following this, in Section 2.2, we turn our attention to causal inference, starting with a high-level overview of a typical causal inference workflow (Section 2.2.1). Then we discuss the fundamentals of causal inference within the context of the potential outcomes framework (Section 2.2.2) before defining the three identifiability conditions necessary for valid causal inferences (Section 2.2.3). We then proceed to discuss two distinct ways one can conduct causal inference: randomised experiments or observational data (Sections 2.2.4 and 2.2.5, respectively). This is followed by a discussion of graphical approaches that assist with the 'identification' step of causal inference (Section 2.2.6). After discussing matters of identification, we consider the second step of causal inference: estimation (Section 2.2.7). This is followed by a discussion of advanced regression modelling methods that can be used to model intricate cause-effect relationships (Section 2.2.8) and a brief overview of machine learning approaches to the estimation of causal effects.

In Section 2.3, we then consider software testing-related research and techniques that involve the use of causal inference. We start with a discussion of the most established line of work that uses causal inference for a software testing-related activity: fault localisation (Section 2.3.1). Following this, we discuss a pair of approaches that focus on *explaining* why a fault or particular software outcome has occurred (Section 2.3.2). We then consider several applications of causal inference to assist software testing-related activities for specific forms of software, such as automotive software (Section 2.3.3), before we introduce what is, to the best of our knowledge, the only general purpose causality-led approach to testing-related activities (other than the one proposed in Chapter 4 of this thesis): Causal Program Dependence Analysis (Section 2.3.4). Then, we discuss two applications of causal inference to observational software engineering data, demonstrating the potential of causal inference to answer causal questions in the software engineering context without necessarily performing experiments (Section 2.3.5). Finally, we conclude our discussion of the intersection between causal inference and software testing by highlighting several key takeaways (Section 2.3.6).

To conclude this chapter, in Section 4.5, we outline the high-level themes, techniques, and topics discussed in this background.

2.1 Software Testing

Software testing is an essential phase of the software development life cycle where the goal is to reduce the risk associated with using software and improve its quality [4]. This can be achieved using a variety of techniques which, at some level, involve executing or analysing software to determine whether it behaves as expected, as defined by some requirements or specification, under a given set of inputs.

Given some requirements, an idealised view of the software testing process may proceed as follows. First, the tester constructs test cases which aim to check whether the software fulfils its requirements. A requirement in this context is a statement of how the software should behave in a particular situation, usually characterised by some set of inputs. A test is then a controlled execution of the software under these inputs that elicits some expected response. The remaining task is to determine whether the actual response matches the expected response. In the affirmative, the test case passes, otherwise it fails. The process used to make this comparison is known as the test oracle.

In this thesis, we are particularly interested in testing computational modelling software. From forecasting the weather to simulating the efficacy of vaccines, computational modelling software is ubiquitous and plays an important role in all of our lives. Given the often critical applications of this class of software, failures in computational models can have catastrophic consequence in terms of both loss of life and fiscal impact. It is therefore of utmost importance that computational modelling software is rigorously tested. However, computational modelling software typically possesses a number of properties that make them particularly challenging subjects to test, including long execution times, a vast and complex input and output space, non-determinism, and exploratory behaviour to name a few.

In this section, we provide an overview of software testing with a particular focus on topics relevant to the testing of complex unwieldy software, such as computational models. To this end, we start by providing an informal overview of the fundamentals of software testing within a seminal software testing framework outlined by Staats et al. [1]. With a grasp of these fundamentals, we then briefly describe two common families of testing techniques - black-box and white-box - before discussing the topic of test adequacy. That is, what constitutes a *good* test suite. After discussing these foundational concepts, we introduce the main challenge that precludes the testing of computational modelling software and motivates this thesis: the test oracle problem [15]. We conclude the section by discussing metamorphic testing [16], a known solution to the test oracle problem [13] and a key topic of this thesis.

2.1.1 An Archetypal Software Testing Framework

Throughout this thesis, we ground our discussion of testing in the software testing framework proposed by Staats et al. [1]. This framework provides a holistic view of software testing that focuses on the interaction between its fundamental components, namely: the program-under-test P, a suite of tests T, a specification S, and a test oracle O. The relationships between these components are summarised in Figure 2.1 below in a modified version of Figure 1 from the original paper [1].

We now describe each feature from Figure 2.1 and provide formal definitions for the targets of software testing: faults, errors, and failures.

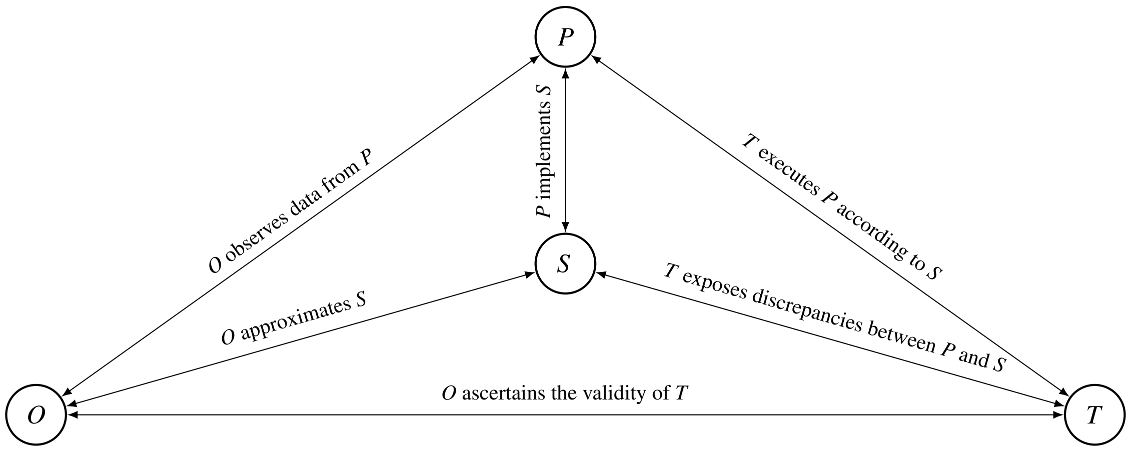

P
P implements S
S
O observes data from P
T executes P according to S
O approximates S
T exposes discrepancies between P and S
O ascertains the validity of T
O
T

Figure 2.1: A graphical representation of the Staats et al. software testing framework [1], highlighting the interrelationships between the program-under-test P, specification S, oracle O, and tests T.

Programs

At a high level, software testing can be described as the practice of executing some program with the aim of finding faults [18]. Thus, central to all forms of testing is some type of target program or system, which is commonly referred to as the **Program-Under-Test** (PUT) or **System-Under-Test** (SUT). To aid our discussion throughout this section, we use the following code example from Figure 2.2, depicting a programs that computes the BMI of a person given their height in centimetres and weight in kilograms according to the formula: BMI = $\frac{\text{weight in kg}}{(\text{height in cm})^2}$. On line two, the height is converted from centimetres to metres before being squared on line three. The weight is then divided by the squared height in metres and rounded to one decimal place on line four to obtain the BMI, which is returned on line five.

```
1 def calculate_bmi(float height, float weight):
2     height_m = height / 100
3     height_m_squared = height_m**2
4     bmi = round(weight / height_m_squared, 1)
5 return bmi
```

Figure 2.2: A program which calculates BMI given a height (cm) and weight (kg) to 1 decimal place.

Tests

In order to determine whether the PUT behaves as expected, it is necessary to construct a series of test cases, often collectively referred to as a test suite, that aim to uncover failures, faults and defects. A test case can be described as an object that defines the expected outputs corresponding to a particular set of input values for the PUT. In addition, when the PUT is a stateful system, a test case may also specify preconditions and postconditions, which describe a particular state the PUT should enter before and after executing the test, respectively.

As an example of a test case for our BMI program in Figure 2.2, we could check whether the program produces the correct output for a given individual with a height of 180cm and weight of 75kg. This should produce a BMI of 23.1. An example implementation of this test case is shown in Figure 2.3, in which we assign the height, weight, and expected BMI as 180cm, 75kg, and 23.1 on lines two, three, and four, respectively, before checking that the program from Figure 2.2 produces the expected BMI given these inputs, on line five.

```
1 def test_bmi():
2     height_cm = 180
3     weight_kg = 75
4     expected_bmi = 23.1
5     assert calculate_bmi(height_cm, weight_kg) == expected_bmi
```

Figure 2.3: A program which calculates BMI given a height (cm) and weight (kg).

Specifications

When testing a program, it is necessary to consider the requirements of the software. These requirements are ideally expressed in some form of *specification*. In the Staats et al. testing framework, a specification is an abstract, ideal representation of correctness of the program with respect to its requirements [1]. Where such a specification is available, the goal of testing can be described as the task of strategically executing the PUT to identify discrepancies between the implementation and the specification. In some situations, however, this idealised version of specification may be unattainable. Instead, the specification often takes the form of a formal or informal artefact that describes the approximate expected behaviour of the PUT. In other words, specifications are often *imperfect* notions of correctness.

Specifications typically express one of two forms of requirement: *functional* or *non-functional* requirements. The former type of requirement concerns the expected functionality of the system. The latter type concerns *how* the PUT performs its function. For example, a functional requirement of the BMI program in Figure 2.2 is that it should compute the correct BMI to one decimal place. A non-functional requirement related to this might be that it should calculate the BMI within a millisecond.

Oracles

Given a program, a specification, and a suite of tests, the remaining challenge is to determine whether the outcomes of tests executed against the program satisfy the specification. That is, whether the output of the test is correct or not. The test oracle is the mechanism used to determine whether the output of a program is correct for a particular test [15]. It achieves this based on some data from the PUT which is often referred to as the oracle data [1] or oracle information [15]. The challenge of providing a test oracle is often referred to as the *test oracle problem*. We discuss this in greater detail in Section 2.1.4. In addition, oracles can be classified further as full or partial test oracles. The former refers to an oracle that characterises the correct response to all inputs, whereas the latter refers to an oracle that characterises the correct response to some inputs.

As an example of a test oracle, consider the following assertion from Figure 2.3, which checks whether the PUT (Figure 2.2) produces the expected output (`expected_bmi = 23.1`) given the test inputs (`height_cm = 180` and `weight_kg = 75`):

```
assert calculate_bmi(height_cm, weight_kg) == expected_bmi
```

This assertion directly specifies the expected outcome as 23.1 given the height and weight of the individual, which are specified as `height_cm = 180` and `weight_kg = 75`, respectively.

Faults, Errors, and Failures

Generally speaking, the goal of a test case is to uncover faults and defects residing in the PUT, which ultimately manifest in failures. We now provide an informal explanation of these terms using the defintions provided by Ammann and Offutt [4]. A *software fault* is some static defect that exists in the software. This is often colloquially referred to as a bug. Typically, faults will produce some incorrect internal state of the software, referred to as a *software error*. Where a software error results in external behaviour that does not satisfy the requirements of the software - that is, the software cannot perform its intended function - we say there is a *software failure*.

```
def calculate_bmi(float height, float weight):
    height_m = height / 1000
    height_m_squared = height_m**2
    bmi = round(weight / height_m_squared, 1)
return bmi
```

Figure 2.4: A faulty program which incorrectly calculates BMI given a height (cm) and weight (kg) to 1 decimal place. The fault originates in the conversion of height in cm to height in m, where the former is divided by 1000 instead of 100.

For example, consider the faulty version of our BMI program shown in Figure 2.4, which contains a fault in the conversion of height from metres to centimetres. Specifically, the programmer has erroneously divided the height by 1000 instead of 100, causing the value of `height_m` to be a tenth of its intended value. Executing our test case from Figure 2.3 against the faulty version of the BMI program (Figure 2.4) will reveal a failure, as the actual result (2314.8) will differ from the expected result (23.1) by a factor of 100. Our faulty program does not correctly compute the BMI.

Conversely, executing the same test case against the original, non-faulty version of the BMI program shown in Figure 2.2 will result in a passing test execution, since the actual result (23.1) matches the expected result. Therefore, in this instance, the test demonstrates that the PUT is able to perform its intended function when supplied with this specific set of inputs.

Now that we have covered the essential components involved in software testing, we discuss two classes of software testing technique: black-box and white-box testing.

2.1.2 Black-Box and White-Box Testing

In general, testing techniques can be divided into two categories: *black-box* and *white-box* techniques. Black-box techniques assume no knowledge of the internal structure of the PUT, including its source code. Instead, the source of knowledge used to drive testing activities such as test case generation (whether manual or automatic) is external to the PUT and often takes the form of some specification or documentation, such as a formal model in *model-based testing* [19]. Conversely, white-box techniques depend on knowledge of the internal structure of the PUT and exploit this to drive the same testing activities. To illustrate the difference between black-box and white-box techniques, let us consider two ways in which we could construct tests for our BMI program (Figure 2.2).

First, suppose we have a requirements document that lists the height, weight, and actual BMI of 50 individuals. We could use this information to design test cases that check whether our program produces the correct BMI for each individual given their height and weight. This is representative of a black-box technique because we are using information that is external to the PUT to drive test case generation.

Second, consider the program in Figure 2.5 that takes a BMI value and places it into one of four weight classes: underweight, healthy weight, overweight, or obese. This program contains one if-statement for each possible weight class, where any individual with a BMI less than 18.5 is classed as underweight, between 18.5 and 25.9 is healthy weight, between 24.9 and 30 is overweight, and anything above 30 is obese.

```
def weight_class(float bmi):
    if bmi < 18.5:
        return "Underweight"
    elif 18.5 <= bmi < 24.9:
        return "Healthy weight"
    elif 24.9 <= bmi < 30:
        return "Overweight"
    elif bmi >= 30:
        return "Obese"

```

Figure 2.5: A program which takes a BMI and places it into one of four weight classes: underweight, healthy weight, overweight, or obese.

On any execution only one of the four branches (that is, code within an if-statement) is executed, depending on the given BMI. Therefore, if a defect is present in a particular branch, the corresponding error will only be realised if the right BMI value is given. It is therefore desirable to construct a test suite that covers all branches. We can achieve this by using the source code or some model derived from it, such as a control flow graph, to identify the different branches and the values of BMI needed to cover them. For example, we could construct a test for each BMI in the set $\{15, 20, 25, 35\}$. This approach to generating test cases is representative of a white-box technique as it exploits the internal structure of the PUT, and is discussed in greater detail in Section 2.1.3.

In this thesis, we primarily focus on black-box testing techniques. This decision is influenced by two factors. First, we focus on testing computational modelling software whose developers and users often lack the expertise we generally assume of a software engineer or testing practitioner [13]. Instead, the expertise of modellers usually lies in the subject matter of the computational model. Second, white-box testing techniques require the tester to have access to, and an understanding of, the inner workings of the software. With these points in mind, we contest that the typical skill-set of a modeller is better-suited to black-box techniques, where testing instead depends on some external source of knowledge or domain expertise that capture how the software should behave.

Now that we have an understanding of both the fundamentals of software testing and two broad approaches to testing in black-box and white-box methods, we turn our attention to the question: What makes a test suite adequate?

2.1.3 Test Adequacy

Testing cannot, and does not, aim to prove the absence of faults and errors [4]. Rather, testing is a discipline in which our goal is to collate evidence that demonstrates whether the PUT fulfils some (sometimes formally specified) requirements, i.e. is fit-for-purpose. So when is our test suite ‘good’ enough? To answer this question, a range of metrics and techniques have been proposed that aim to measure the *adequacy* of a test suite. In addition to appraising the quality of an existing test suite, test adequacy measures also provide a target or stopping criterion for automatic test generation methods [20, 4].

We now discuss three dominant approaches to measuring test adequacy: structural coverage criteria, data coverage criteria, and mutation analysis.

Structural Coverage Criteria

The first form of test adequacy is driven by the observation that a good test suite should exercise a wide range of the PUT’s behaviour. This idea is captured by a family of metrics known as structural coverage criteria which measure the proportion of a program or its specification that is executed (covered) by the test suite.

In white-box testing, structural coverage criteria concern the coverage of some structural feature of the PUT, including statements, branches, and paths [21]. More specifically, statement coverage measures the proportion of statements in the PUT that are covered by a test suite; branch coverage measures the proportion of possible branches of switch-cases, if-then-elses, do-whiles etc. [18] exercised by a test suite; and path coverage measures the proportion of execution paths in the control flow graph that are covered by a test suite. These coverage criteria are typically used as targets for test case generation, with the aim of generating a test suite that achieves a sufficient level of statement, branch, or path coverage.

Although structural coverage criteria is most often associated with the source code of the PUT, analogous coverage criteria can also be defined for certain forms of model-based specification that express some form of control-flow, such as models expressed in UML [22]. In addition, model-specific structural coverage criteria have been defined, such as all-states, all-configurations, and all-transitions for **Finite State Machines** (FSMs) [20].

Data Coverage Criteria

Alternatively, we can measure the adequacy of a test suite with respect to its coverage of the input space of the PUT. This is quantified by data coverage criteria. In an ideal world, a test suite should cover all possible combinations of input values, referred to as *all-combinations* coverage. However, for most forms of software, there are too many possible combinations to exhaustively test.

Take our BMI program from Figure 2.2 for example. Even if we discretise the weight and height input variables to integer values in the ranges [10, 300) and [30, 250), respectively, there are $290 \times 220 = 63800$ possible input pairs to test. In general, if there are m inputs each with a domain consisting of k possible values, there are k^m combinations to test. Thus, the number of possible combinations grows exponentially with the number of inputs. This phenomenon is referred to as combinatorial explosion. One approach to minimise combinatorial explosion is to weaken our target coverage criterion to *pairwise* or *n-wise* coverage [20]. As the names suggest, these coverage critieria consider all possible combinations of parameters of size n ($n = 2$ in the case of pairwise) and can be satisfied using significantly fewer tests.

Alternatively, we can seek to reduce the effective size of the input space by employing techniques such as equivalence partitioning and boundary value analysis. The underlying idea of equivalence partitioning is to divide the input domain into regions where all input values are considered as equivalent. In this context, two input values are considered to be equivalent if we expect that if one reveals a fault, the other would too [18]. For example, for a numerical program that takes the square root of a numerical input, we might partition its input domain into NEGATIVE, ZERO, and POSITIVE. However, once we have identified equivalence partitions, an open question is which input(s) should we select?

To this end, boundary value analysis extends equivalence partitioning based on the conventional wisdom that faults tend to occur on the boundaries of equivalence partitions. Instead of selecting any value from an equivalence partition, in boundary value analysis, we select values from the edges of equivalence partitions that are, at least intuitively, more likely to expose a fault. For example, if our numerical program were implemented in Java, we might select input values of 0.0001 and `Integer.MAX_VALUE` for covering the POSITIVE equivalence partition. These extremes of the POSITIVE class could, for example, expose rounding errors that do not meet the required level of precision.

Mutation Testing

Until now, we have focused on measures of test adequacy that consider coverage of either the PUT or an artefact derived from it. Conceptually, these measures of test adequacy indicate how broad or narrow our test suite is in terms of the range of behaviours it tests. An alternative dimension of test adequacy is, of course, how effective a test suite is at detecting faults i.e. what proportion of the total faults does the test suite detect? However, the problem of determining whether all faults have been detected in a program is undecidable [4]. Therefore, given some test suite, we cannot ordinarily quantify the proportion of bugs that it has detected.

Sidestepping this undecidability problem, *mutation testing* provides an alternative method for measuring test adequacy that involves seeding artificial faults into the PUT and measuring the ability of a test suite to detect them. The first step of mutation testing is to apply a series of *mutation operators* which manipulate the syntax of the PUT, forming so-called *mutants*. There are many different types of mutation operator that are generally applicable, as well as language specific operators. We provide a summary of common classes of mutation operators, as outlined by Ammann and Offutt [4], in Table 2.1.

Operator	Description	Pre-Mutation	Post-Mutation
Absolute Value Insertion (AVI)	Replace an arithmetic expression with its positive or negative absolute value or a function that tests if the expression evaluates to zero.	`x = 2*z`	`x = abs(2*z)`
Arithmetic Operator Replacement (AOR)	Replace occurrences of arithmetic operators with different arithmetic operators.	`x = 2*z`	`x = 2+z`
Relational Operator Replacement (ROR)	Replace occurrences of relational operators with different relational operators.	`if (x > 2)`	`if (x == 2)`
Conditional Operator Replacement (COR)	Replace occurrences of logical operators with other conditional operators.	`if (x || y)`	`if (x && y)`

Table 2.1: A selection of common mutation operators outlined by Ammann and Offutt [4].

Once a mutant has been formed, the original PUT and the mutant are then executed against the same test suite. We say that the mutant is *killed* by some test from the test suite if the test passes on the PUT, but

```python
def f(n: int):
  if n <= 1:
    return n
  else:
    return f(n-1) + f(n-2)
```

(a) The original program.

```python
def f(n: int):
  if n <= 1:
    return n
  else:
    return f(n-1) * f(n-2)
```

(b) The corresponding mutant.

Figure 2.6: A recursive implementation of a function for obtaining a Fibonacci number and a mutant obtained by applying the arithmetic operator replacement mutation operator, swapping + for * on line 5.

fails on the mutant. Otherwise, it is said to be *alive*. After executing the test suite against all mutants, we compute the mutation score (MS) as our measure of test adequacy, which is the proportion of killed mutants (M_k) to total mutants (M_t):

$$\text{MS} = \frac{M_k}{M_t} \tag{2.1.1}$$

In some cases, a mutation operator can create a mutant that is semantically equivalent to the original program. In this case, we refer to the mutant as an *equivalent mutant*. Ideally, equivalent mutants (M_e) should be removed from mutation testing and the computation of the mutation score:

$$\text{MS} = \frac{M_k}{M_t - M_e} \tag{2.1.2}$$

However, in general, detecting equivalent mutants is an undecidable problem [23] and manual detection and removal of equivalent mutants is time consuming [24]. To this end, several approaches have been proposed that help to to alleviate the equivalent mutant problem, such as higher order mutation testing [25, 23] and the use of program-slicing [26]. Nonetheless, even without removing equivalent mutants, the naive mutation score from Equation (2.1.1) still provides a useful indicative measure of the fault detection capability of a test suite [23].

While mutation testing is most often associated with programs (i.e. mutating source code), it can also be applied to formal specifications, such as FSMs. This approach, originally introduced by Budd and Gopal [27] who applied it to predicate calculus specifications, is referred to as specification-based mutation and proceeds similarly to program mutation. First, a series of mutation operators are applied to the specification. For example, Fabbri et al. defined nine separate mutation operators specific to FSMs that manipulate the states, transitions, and outputs of the FSM, such as transition deletion (arc-missing) [28]. Then, to kill the mutant specification, a sequence of states that is compatible with the original FSM, but incompatible with the mutant FSM, must be identified. This has a direct correspondence to a mutant-killing test.

One of the advantages of this formulation of mutation testing is that, given the finite domain imposed by a specification, such as an FSM, and a property expressed in temporal logic, model checkers can be used to automatically generate *counterexamples* [4]. That is, a sequence of states that is accepted by the original state machine, but rejected by the mutant. As stated above, this is essentially a mutant-killing test. Furthermore, the finite domain imposed by the specification makes the equivalent mutant problem decidable, enabling model checkers to facilitate automatic detection of equivalent mutants [29].

In this context, equivalent mutants manifest as mutated specifications for which the model checker cannot provide a counterexample.

Now that we have discussed measures of test adequacy that can be used to guide test generation and appraise the quality of existing test suites, in the following subsection, we turn our attention to the natural next step: establishing whether the output of some test case is correct or incorrect - *the test oracle problem* [15].

2.1.4 The Test Oracle Problem

As discussed previously in Section 2.1.1, the test oracle is the mechanism used to determine whether the output of some test case is correct or incorrect [30]. The challenge of providing this artefact is known as *the test oracle problem*. However, the extent to which this problem hinders testing varies greatly from system to system. For some systems, the test oracle problem is trivial, while for others it is an arduous and time consuming task. Worse yet, for some classes of software, test oracles are fundamentally or pragmatically unattainable [30]. Weyuker branded this class of software as *non-testable*. In this subsection, we briefly discuss so-called 'non-testable' software, focusing on three classes of software that are particularly susceptible to the test oracle problem, before describing four widely recognised categories of test oracle, some of which provide a solution for even non-testable systems.

Non-Testable Software

A program is said to be non-testable if either: (1) an oracle does not exist or (2) it is theoretically possible but highly impractical to provide one [30]. Such a program is referred to as being non-testable because, without an oracle, we possess no means to appraise the output, making any testing efforts futile. Weyuker noted three classes of program that are non-testable by nature[1]:

1. Exploratory programs: By exploratory, we refer to programs which are written to answer, or predict an answer, to a previously unanswered question. Of particular relevance to this thesis, computational models generally fall within this class of software. For example, epidemiological models are often used to evaluate prospective interventions, such as vaccines or social policies, by simulating their impact on future outcomes such as disease progression and mortality rates. At the time of simulation, the correct outcome of the simulated scenario is fundamentally unknowable. It is only after the event has actually occurred that we could possibly elicit an oracle.

2. Programs with unwieldy output spaces: Some classes of software have such a vast and complex output space that makes providing a suitably accurate oracle infeasible. Instead, a human must often 'eyeball' the outputs [30]. Again, some forms of computational model fall within this class of software. For example, Covasim — a COVID-19 agent-based model that we use as a motivating example in Chapter 3 — has over 50 outputs, many of which are complex time series representing viral dynamics. For any class of software with such an expansive output space, attempting to characterise correctness over all outputs would be highly impractical.

3. Complex programs: It is not uncommon for a program to be so complex in nature that a tester misinterprets its intended behaviour. This complexity may arise from either the subject-matter of the software or the implementation itself. A common source of complexity that is problematic for

[1]In the original paper [30], Weyuker did not provide a name for the three classes of program, but described the details of them. We therefore selected these labels to best reflect the descriptions given.

testing is non-determinism, where programs contain decision points at which some execution path is chosen arbitrarily, leading to inconsistent test outputs. Once again, given the complexity of their subjects and often non-deterministic nature, computational models provide an apt example of this class of software [13].

While non-testable programs have characteristics that, in the worst case, make it fundamentally impossible to specify the correct output corresponding to some input, a number of approaches have been proposed that provide a *partial test oracle*. That is, an oracle that can ascertain the correctness of some but not all test cases [15]. In addition, Weyuker introduced the notion of a *pseudo-oracle*: a reference implementation of the PUT that can act as a basis for comparison. Although there is no guarantee that this reference implementation is itself correct, it can help to identify output discrepancies that may be representative of errors or developer misunderstandings.

In recent years, the theory surrounding test oracles has been developed greatly, culminating in four widely recognised, distinct forms of oracle [15]: specified, derived, implicit, and human. We now discuss these forms of oracle in turn and the costs associated with the latter human oracles.

Specified Test Oracles

Specified test oracles use some form of formal or informal specification as oracle data. Such specifications often take the form of formal models expressed in modelling languages like Z [31] and VDM [32]. This form of specified test oracle carries the advantage that, when used as part of a model-based testing approach, the underlying model can act as both a guide for generating an adequate test suite (as discussed in Section 2.1.3) as well as an automated test oracle. For example, state machine diagrams can be used to generate tests that use state invariants as test oracles [33]. In addition, assertions built into conventional programming languages, like `assert` in Java and Python, are also considered to be specified test oracles. Indeed, in our example test case for our BMI program Figure 2.3, we used an assertion as our test oracle.

Derived Test Oracles

Derived test oracles use some information or properties derived from artefacts of the PUT or other implementations of the PUT, such as program executions. One example of a derived test oracle is a *metamorphic relation*; a property defined over the inputs and outputs of a program that captures the expected effect of a prescribed change to the inputs on the outputs. The process of constructing and executing metamorphic relations against a program is known as *metamorphic testing* and is one of the key topics of concern in this thesis. We provide an in-depth background into metamorphic testing in Section 2.1.5. Alternatively, reference implementations can act as a form of derived oracle in testing activities such as regression testing, where the goal is to determine whether code changes have in some way regressed the quality of a previous version (e.g. introduced faults or reduced code coverage). Here, the previous version of the PUT acts as an oracle. This form of test oracle is also referred to as a *pseudo-oracle*.

Implicit Test Oracles

Implicit test oracles use widely accepted knowledge surrounding the PUT to differentiate correct and incorrect outcomes. For example, if a program crashes unexpectedly for a given input, it is generally accepted that this can be attributed to some fault or defect. This constitutes an implicit test oracle as we can conclude that, given this input, the behaviour of the PUT is abnormal without requiring additional work or information. Rather, we lean on the implicit assumption that if the program has crashed unexpectedly, there is some underlying fault. However, it is important to note that what is considered to be implicit knowledge in one domain is not necessarily implicit to another. Therefore, there is always some ambiguity in establishing what knowledge is considered implicit, given the context of the PUT.

Human Test Oracles

Where no specification, derived artefact, or implicit knowledge is available to use as oracle information, as a last resort, a human can act as a test oracle. In practice, this means the task of determining the correctness of a program is left to human intuition [34]. Note that, while similar in principle, human and implicit oracles are differentiated by one key factor: human oracles depend on domain expertise or specialist knowledge rather than some generally accepted truth. Human oracles are commonplace in the domain of computational modelling software, where scientists often resort to professional judgement to assess the validity of model behaviour [13]. For example, in scientific domains, computational models are often developed that produce complex graphical outputs. The expected shape or characteristics of such graphical outputs are often known to domain experts but may be difficult to formally specify or characterise. Instead, a domain expert will typically ‘eyeball’ the results. While this is a commonplace approach, it relies on domain expertise that may be difficult to communicate to non-domain experts such as testers and software developers.

Oracle Cost

In situations where a full or partial oracle is unavailable and we instead rely on human input, there is a need to reduce the amount of manual effort required to judge the correctness of software. This is referred to as the human oracle cost [35] and is often separated into sub-types: the quantitative and qualitative costs.

The quantitative cost refers to the cost associated with test execution, which is predominantly mediated by two factors: (1) the complexity of individual test cases, and (2) the size of test suites. Conceptually, a test case is considered to be complex if, for example, it has many method calls, some of which are not relevant to the goals of testing. Similarly, a test suite is considered to be unwieldy if it contains many test cases, which may include test cases that do not achieve the goals of testing (e.g. coverage criteria, see Section 2.1.3). Thus, in order to reduce the quantitative cost, it is necessary to reduce redundancy in test cases and test suites. To this end, a common solution is to employ techniques that reduce the size of test cases and test suites [15], such as test suite reduction techniques [36]. Furthermore, the use of advanced test case generation techniques can help to reduce the number of test cases in comparison to randomly generated tests.

The qualitative cost concerns the ease of which test cases and testing activities can be understood by a human. When testing techniques rely on randomly generated inputs, for example, there is an additional burden placed on the tester to interpret the scenario prescribed by the inputs [15]. If, instead, domain knowledge is incorporated into the test generation process, the tests are naturally easier to comprehend as the scenario prescribed by the inputs and the corresponding output are more likely to be known. In general, the qualitative cost associated with the test oracle problem can be alleviated by leaning on domain knowledge to construct more comprehensible test cases. To this end, McMinn et al. proposed a series of techniques that incorporate human knowledge into the test generation process to make automatically generated test inputs more interpretable, reducing the qualitative human oracle cost as a result [35].

Now that we have considered non-testable programs and the various forms of test oracle, as well as the costs associated with human oracles, we discuss metamorphic testing: a known solution to the test oracle problem and one of the primary topics of concern in this thesis.

2.1.5 Metamorphic Testing: A Solution to the Test Oracle Problem

Metamorphic testing [37] is a known solution to the test oracle problem [38]. Instead of testing software by asserting the expected output of an individual execution, metamorphic testing effectively sidesteps the test oracle problem by asserting the expected *effect* of a particular transformation made to the input space on the output space. In some respects, this can be perceived as a way of reducing the qualitative human oracle cost: the domain expert no longer needs to know a precise output corresponding to some input, but the expected effect (e.g. an increase, decrease, no change, or specific change) of some change (e.g. doubling an input).

As a solution to the test oracle problem, metamorphic testing has been applied in a myriad of different domains, ranging from phylogenetic inference [39] to autonomous vehicles [40]. Such applications of metamorphic testing have proved to be successful, uncovering numerous bugs that are ordinarily difficult to detect without access to a full test oracle. For example, Donaldson et al. [41, 42], proposed a metamorphic testing approach and accompanying tool, GLFuzz, that aims to detect bugs in graphics shader compilers by applying semantics-preserving transformations. In a large scale experiment, this tool detected more than 60 graphics compiler bugs, including security-critical vulnerabilities affecting WebGL.

In this sub-section, we introduce the fundamentals of metamorphic testing and discuss solutions to two notorious limitations surrounding metamorphic testing - namely: the construction of metamorphic relations and handling non-determinism. We then provide an example application of metamorphic testing to so-called non-testable software.

Fundamentals

In order to introduce the fundamentals of metamorphic testing, we turn our attention to a simple property of the sine function: $sin(x) = sin(\pi - x)$. In the context of metamorphic testing, this property is an example of what is referred to as a *metamorphic relation* and is defined over two distinct executions of the PUT: the *source test case* $x_s = x$, and the *follow-up test case* $x_f = \pi - x$. Note that the follow-up test case is formed by applying a purposeful transformation to the source test case which, in this case, is subtraction from π. Given this intuition, we formally define a metamorphic relation as follows[2]:

Definition 2.1 (*Metamorphic Relation*)

A *metamorphic relation* $\mathcal{MR} = (x_s, T, R)$ is a functional property of the program-under-test, P, that captures how a *source test case* x_s can be transformed into a corresponding *follow-up test case* $x_f = T(x_s)$ in such a way that some property R over the outputs of these test cases can be anticipated, such as equality: $P(x_s) = P(x_f)$. Here, T is referred to as a transformation or generation mechanism and is a function $T : x_s \rightarrow x_f$ that transforms the source test case into its

[2]This definition is motivated by the informal description of a metamorphic relation given by Segura et al. in their survey on metamorphic testing [43]. However, we use slightly different notation here and, to avoid redundancy, do not include the follow-up test case as its own component since it is generated by applying the transformation T to x_s.

corresponding follow-up test case.

Notice that metamorphic relations provide built-in 'recipes' for generating follow-up test cases from source test cases. In our example, this corresponds to subtracting the source test case from π. Hence, our transformation is the function $T : x \rightarrow \pi - x$. In many cases, such as this example, the underlying property R of the metamorphic relation is sufficiently simple that the transformation function is implicit, meaning that the property itself provides a succinct description of the metamorphic relation. For this reason, the literature commonly refers to the underlying property, such as $sin(x) = sin(\pi - x)$, as the metamorphic relation. We also follow this convention but use the formal notation from Definition 2.1 where the transformation mechanism is not implicit.

The process of applying metamorphic testing typically proceeds via the following steps [44, 43], where $P(x)$ denotes execution of the PUT with test case x:

1. Identify a property of the PUT that, implicitly, captures the expected effect of some transformation to the input space.

2. Generate a series of source test cases $X_s = \langle x_{s1}, x_{s2}, \ldots, x_{sn} \rangle$ and execute them to obtain their corresponding outputs $\langle P(x_{s1}), P(x_{s2}), \ldots, P(x_{sn}) \rangle$.

3. Define the (until now) implicit generation mechanism T and apply this to the series of source test cases X_s to obtain the corresponding follow-up test cases $X_f = \left\langle x_{f1}, x_{f2}, \ldots, x_{fn} \right\rangle$.

4. Execute the generated follow-up test cases X_f to obtain their outputs $\left\langle P(x_{f1}), P(x_{f2}), \ldots, P(x_{fn}) \right\rangle$.

5. Iterate over the series of source and follow-up output pairs and determine whether they satisfy the specified relation. For example, if we are testing equality: $P(x_{s1}) = P(x_{f1}) \wedge P(x_{s2}) = P(x_{f2}) \wedge \ldots \wedge P(x_{sn}) = P(x_{fn})$.

To further demonstrate this process, let us consider each step in the context of our sine example:

1. Identify the underlying property of the metamorphic relation: As described above, $sin(x) = sin(\pi - x)$.

2. Generate the source test cases: Given that the domain of sine is the real numbers, we can sample the inputs for our source test cases at random. For example, supposing we randomly sample three values to generate three source test cases: $X_s = \langle 0.5, 10, -2 \rangle$.

3. Define the generation mechanism and apply it to the source test cases: We need to subtract our source test cases from π, which can be achieved using the function $f(x) = \pi - x$. Applying this to each source test case yields the following series of follow-up test cases: $X_f = \langle \pi - 0.5, \pi - 10, \pi + 2 \rangle$.

4. Execute the pairs of source and follow-up test cases and check whether they satisfy the property: By considering the three pairs of source and follow-up test outputs, we need to satisfy: $sin(0.5) = sin(\pi - 0.5) \wedge sin(10) = sin(\pi - 10) \wedge sin(-2) = sin(\pi + 2)$. Assuming the property is correctly specified and the previous steps are implemented correctly, then if any of these relations fail, our implementation of sine contains a fault.

Metamorphic Relation Construction

A key barrier to the application of metamorphic testing lies in the construction of metamorphic relations [44]. This is a notoriously challenging task as it depends heavily on the tester's intuition or on the availability of an expansive and established specification [45] to identify suitable properties of the PUT Thus, in general, this process is time-consuming and heavily dependent on domain expertise in the subject-matter of the PUT [43]. To address this problem, a number of guidelines [46, 47] and techniques have been developed that assist the process of constructing metamorphic relations. Most of these techniques can be grouped into four main themes: composition of existing metamorphic relations, detection of pre-defined metamorphic relations, discovery of novel metamorphic relations, and specification-derived metamorphic relations.

Metamorphic Relation Composition Based on the observation that existing metamorphic relations can often be 'chained together' to form new, potentially more effective ones, Liu, Liu, and Chen developed an approach known as *Composition of Metamorphic Relations* (CMR) [48]. This approach involves identifying so-called 'compositable' metamorphic relations, where a metamorphic relation MR_y is said to be compositable to MR_x if and only if, for all source test cases of MR_x, the corresponding follow-up test cases are valid source test cases for MR_y. Through an experimental study, the authors found that composite metamorphic relations tend to be at least as effective as their constituent relations at detecting faults and sometimes even better. In addition, they found that composite metamorphic relations are generally more cost effective than testing the constituent relations themselves. However, this approach still requires the user to construct the initial metamorphic relations and to determine whether they are compositable which, for complex input spaces, may be non-trivial.

Detecting Pre-Defined Metamorphic Relations The second form of approach for metamorphic relation construction involves predicting or detecting whether some pre-defined series of metamorphic relations hold for a given program. Strictly speaking, this approach does not involve the construction of metamorphic relations, but provides a means for identifying *likely* metamorphic relations i.e. those that probably hold for the PUT. This approach was first developed by Kanewala and Bieman [49] for numerical functions, who framed the problem as a supervised machine learning task, in which supervised machine learning algorithms, such as support vector machines, are trained on features of a control flow graph extracted from the PUT and used to predict whether a selection of pre-defined metamorphic relations should hold.

Similarly, Su et al. [50] developed Kabul, a technique which dynamically instruments the PUT to collect execution profiles that, roughly speaking, monitor the output state of the PUT when executed under a variety of different inputs. Using the collected data, the proposed approach uses a series of 'output state checkers' that look for pre-defined changes in the output space, such as a constant additive offset, that can be attributed to a particular change in input. While both approaches have been shown to be relatively effective at detecting metamorphic relations with good fault detection capabilities, they both require a priori knowledge of some likely metamorphic relations.

Metamorphic Relation Discovery The third class of approach concerning the construction of metamorphic relations is metamorphic relation discovery. Unlike the previous approaches, this class of approach does not depend on previous knowledge of metamorphic relations that either hold or may hold in the PUT and instead 'discovers' metamorphic relations without a priori targets. For example, Zhang et al. [51] developed a search-based approach to detect metamorphic relations that can be expressed as linear or quadratic polynomial equations. This approach uses particle swarm optimisation to detect metamorphic relations that hold for most input values of the input space. Since exhaustive testing of the input space is generally infeasible or even impossible, the approach cannot guarantee that the resulting metamorphic relations hold for all inputs.

In a similar vein, Zhang et al. [52] developed an approach for automatically discovering metamorphic relations, called AutoMR. Unlike the search-based approach of Zhang et al., AutoMR can infer inequality polynomials and can generalise beyond linear and quadratic polynomials. Furthermore, this approach contains an additional step that removes redundant metamorphic relations, such as highly similar relations. While both approaches are effective at discovering metamorphic relations, they are limited to a very particular class of program: deterministic programs with numerical inputs.

All three of the aforementioned approaches share a fundamental limitation in their concept: they depend on detecting, predicting, or inferring metamorphic relations from the PUT itself. Thus, the resulting metamorphic relations test the program against its actual behaviour as opposed to its intended behaviour (i.e. the specification). This is problematic as it realistically limits the technique to detecting metamorphic relations that hold in the PUT. By contrast, unless detected by chance as false positives, metamorphic relations that should hold but do not hold would not be detected. Nonetheless, this approach is still suitable for certain classes of testing, such as regression testing [52].

Specification-Derived Metamorphic Relations An alternative approach which does not rely on constructing metamorphic relations based on the PUT itself is METRIC, proposed by Chen et al. [45]; a specification-led approach built upon the category-choice framework. In this framework, the user specifies a series of categories (key parameters) and choices (notable values for each category) that ultimately form test frames, which are disjoint partitions of the input space. These test frames are essentially abstract test cases covering valid combinations of choices and support automatic generation of concrete test cases.

In order to construct metamorphic relations, METRIC presents users with distinct pairs of test frames identified from the specification and asks them to consider whether the difference in input between them would cause a change in output that can be anticipated. If so, the user has successfully identified a metamorphic relation for which the source and follow-up test cases can be automatically generated. While this approach does not encounter the same fundamental limitation of the previous approaches, it only partially automates the construction of metamorphic relations. Specifically, it reduces the burden placed on the user by automatically presenting them with the possible changes in input (i.e. T in Definition 2.1). Furthermore, the user must still construct the underlying specification.

Statistical Metamorphic Testing

Until now, we have confined our discussion of metamorphic testing to deterministic programs: programs that will produce the same output if executed with the same inputs multiple times. However, as discussed in Section 2.1.4, non-testable software [30] and non-determinism often go hand in hand. Therefore, for metamorphic testing to be considered a widely viable solution to the test oracle problem, it must accommodate for non-determinism. This limitation in the conventional formulation of metamorphic testing was identified by Guderlei et al. [53] who proposed its extension to non-deterministic contexts, known as **Statistical Metamorphic Testing** (SMT).

Where metamorphic testing typically concerns a pair of *individual* test executions (one for the source test case and one for the follow-up), SMT concerns a pair of *distributions* of test executions obtained by repeatedly executing the source and follow-up test cases. Consequently, a statistical metamorphic relation must specify the expected change in output in terms of some characteristic or property of the output distribution. These statistical properties can be readily tested using statistical hypothesis testing. For example, for a non-deterministic system that produces a numerical output, we might expect the mean value to increase if we change a Boolean input, I, from 0 to 1. Using SMT, we could test this property as follows:

1. Execute the PUT numerous times - say 30 - with both the source test case ($I = 0$) and the follow-up test case ($I = 1$) to obtain corresponding output distributions.
2. Calculate and contrast the mean value of each distribution.
3. Apply an appropriate statistical hypothesis test, such as a student's t-test, to determine if there is a statistically significant difference between the two means.

Hence, SMT is a relatively simple extension of metamorphic testing that takes the underlying concept and generalises it to non-deterministic systems through the application of statistical hypothesis tests to output distributions. However, this approach has one major limitation. It increases the cost associated with metamorphic testing as the source and follow-up test cases must be executed enough times to provide an adequate sample size for the statistical hypothesis test. For classes of software like computational models that often have significant execution times, this presents a significant limitation. In Chapter 6, we mitigate this limitation by employing causal inference methods to predict test outcomes from existing, potentially confounded test data.

Metamorphic Testing of Non-Testable Software: An Example

Given that this thesis focuses on computational modelling software, which, as discussed in Section 2.1.4, is a form of non-testable software [30], we now discuss an example application of metamorphic testing to non-testable software in the domain of computational biology.

In the field of computational biology, researchers are often concerned with establishing the evolutionary history of various organisms, from cancerous cells to animal species. This practice is known as phylogenetics and generally involves the use of phylogenetic inference techniques that infer phylogenetic trees from a series of specified DNA sequences, which are trees representing evolutionary history. This practice has important applications, such as identifying disease variants in an epidemic so that targeted and effective interventions can be designed and deployed [39].

However, phylogenetic inference programs are notoriously difficult to test and validate as, in most practical use cases, the 'true' phylogenetic tree is unknown. Therefore, this class of software can be considered exploratory as per our discussion of non-testable software [30] in Section 2.1.4. Even for non-exploratory applications, the only way to verify the output is to work through the underlying algorithms by hand, which, for all but the most trivial inputs, is infeasible [54]. Therefore, phylogenetic inference programs are a good example of software that suffers with the test oracle problem.

To address this problem, Sadi et al. [54] proposed seven novel metamorphic relations to facilitate the application of metamorphic testing to phylogenetic inference programs. These programs take a matrix as input, where each row is a DNA sequence described by a sequence of the four nucelotides 'A', 'B', 'C', or, 'D', and each column of this matrix is referred to as a 'site'. Broadly speaking, the proposed metamorphic relations all manipulate the input matrix in a principled way that should affect the output tree in a manner that can be anticipated.

As an example, let us consider two of the proposed metamorphic relations that operate on the input matrix. First, MR_{swap} takes two sites (columns of the matrix) and swaps them, with the expectation that the resulting tree remains unchanged. Second, MR_{copy} concatenates a copy of the original matrix to itself, which should produce a tree which is twice the length of the original. These metamorphic relations are illustrated in Equation (2.1.3) below, where the MR_{swap} has swapped the 2nd and 5th columns (sites), specifically.

$$\begin{bmatrix} A & A & C & G & T & A & A & C & G & T \\ A & A & T & G & G & A & A & T & G & G \\ A & A & T & G & G & A & A & T & G & G \end{bmatrix} \xleftarrow{MR_{copy}} \begin{bmatrix} A & T & C & G & A \\ A & G & T & G & A \\ A & G & T & G & A \end{bmatrix} \xrightarrow{MR_{swap}} \begin{bmatrix} A & A & C & G & T \\ A & A & T & G & G \\ A & A & T & G & G \end{bmatrix} \tag{2.1.3}$$

Using mutation analysis (see Section 2.1.3), the authors evaluated the fault detection capability of the proposed metamorphic relations on three phylogenetic inference programs, seeding a series of artificial faults to form mutants. Metamorphic testing was then applied to these mutant programs, where the source test cases were generated from both real-world data and randomly generated DNA sequences. Then, metamorphic relations such as those depicted in Equation (2.1.3) were applied to the source test cases to obtain corresponding follow-up test cases. This resulted in a total of 70000 source and follow-up test case pairs that were subsequently executed and their respective outputs contrasted in accordance to the expected outcome of the metamorphic relationship.

The results of mutation analysis showed that across all three programs, the metamorphic relations were highly effective, with all mutants being killed by at least one metamorphic relation. In addition, the authors reported the proportion of the source and follow-up test pairs that revealed a fault, which ranged from 25.15% to 49.96% across the programs, with little difference between those generated randomly and those taken from real-world data. Therefore, the proposed metamorphic relations provide an effective way to detect faults in the phylogenetic inference programs without having to know the precise correct output, alleviating the test oracle problem.

Furthermore, this paper was used as the basis for a case study of the CMR approach to metamorphic testing proposed by Liu, Liu, and Chen [48] that we introduced earlier in this sub-section. In their case study, Liu, Liu, and Chen composed various combinations of the matrix-based metamorphic relations, including those shown in Equation (2.1.3). The authors found that, in general, the composite metamorphic relations were more effective at detecting faults than their constituent parts, while requiring fewer test executions. Hence, CMR can be applied to phyologenetic inference programs to effectively generate new composite metamorphic relations while reducing the associated cost.

2.1.6 Summary

In this section, we have reviewed key concepts and topics from software testing that are relevant to this thesis. We started by introducing the fundamental components involved in software testing through the Staats et al. software testing framework [1]. Later, in Chapter 4, we use this framework as the basis to develop our own software testing framework that incorporates causal inference directly into the testing process. We then gave a brief overview of white-box and black-box testing techniques before discussing measures of test adequacy, including various coverage criteria and, of particular relevance to

Chapter 5, mutation testing. After introducing these fundamental concepts, in Section 2.1.4, we turned our attention to two closely related subjects that are key motivating factors behind this thesis: the test oracle problem and Weyuker's notion of non-testable software [30]. In Chapter 3, we study two example computational models to demonstrate what non-testable software looks like in practice from a testing perspective. Finally, we introduced metamorphic testing, a tried and tested solution to the test oracle problem that is central to all remaining chapters of this thesis.

2.2 Causal Inference

A goal that is common to most, if not all, fields of science is to answer questions that are causal in nature. In epidemiology, for example, epidemiologists often ask questions such as *What is the effect of a vaccine on morbidity rates in the population?* Similarly, economists might ask *Had we used policy A rather than policy B, would the economy still fall into recession?* In order to answer such questions, we require a careful analysis of the cause-effect relationships that govern data generation in order to design experimental or statistical procedures that appropriately isolate and measure the cause-effect relationship of interest. It is this process of measuring and estimating causal effects in the pursuit of answers to such causal questions that is the objective of *causal inference* [55].

In this thesis, our high-level objective is to understand how causal inference can be leveraged within a metamorphic testing context and, in particular, whether the advantages of this methodology translate to the setting of computational modelling software. To this end, in Chapters 3 and 4, we formalise metamorphic testing as a problem of causal inference and define a causal inference-driven software testing framework that facilitates the direct application of causal inference to causality-led software testing problems.

With this in mind, this section has two broad objectives. First, we want to provide the reader with an intuition for the overarching goals and purpose of causal inference and, critically, the nuances that differentiate it from conventional statistical approaches. For example, in Section 2.2.3, we explain the conditions that must hold for an estimate to be considered 'causal'. Second, we introduce the definitions and nomenclature necessary to understand later chapters of the thesis. Throughout this section, we focus on the potential outcomes framework [56] and Pearl's graphical interpretation of this framework [57]. Our primary reference for most definitions and concepts in this section is Hernán and Robins' textbook: *What if?* [58]. We also draw upon Pearl's *Causality* [59] for content related to graphical causal inference, particularly in Section 2.2.6.

The remainder of this section is structured as follows. We begin with a brief high-level overview of the workflow of causal inference in Section 2.2.1, before formally defining the fundamentals in the potential outcomes framework [56] in Section 2.2.2. We then define and discuss the conditions necessary to make valid causal inferences in Section 2.2.3 and describe the two dominant approaches to causal inference in Sections 2.2.4 and 2.2.5. In Section 2.2.6, we turn to Pearl's graphical approach to causal inference, before explaining methods for estimating causal effects in Section 2.2.7, including regression modelling techniques in Section 2.2.8 and a brief discussion of machine learning techniques in Section 2.2.9.

2.2.1 A High-Level Overview of Causal Inference

Broadly speaking, causal inference can be broken down into two distinct tasks that collectively aim to answer some causal question. We explain these two steps in the two paragraphs that follow and summarise the typical causal inference workflow in Section 2.2.1.

The first of these tasks is commonly referred to as *identification* and concerns the design of experimental or statistical procedures that, with infinite and perfect data, are capable of estimating some causal quantity of interest, known as the causal estimand. The causal estimand can be considered as the target quantity that, once estimated, permits us to answer our causal question. With this in mind, the key challenge of identification is to ensure that a number of *identifiability conditions* are met that allow us to endow any resulting estimate with a *causal* interpretation. We discuss these identifiability conditions in detail in Section 2.2.3 and explain the nuances of these conditions within the context of two different approaches to causal inference: randomised experiments in Section 2.2.4 and statistical procedures with observational data in Section 2.2.5.

The second problem is often referred to as *estimation* and involves estimating the value of an identified estimand (i.e. the output of identification). Put simply, the goal of estimation is to compute or estimate the causal effect of interest. For most non-trivial causal questions, this involves using well-known existing statistical techniques that are capable of 'adjusting for' various forms of bias in the data, such as standardisation. We focus on two approaches to estimation: (1) how causal effects can be estimated directly with perfect and infinite data, and (2) how statistical models can be used in situations where data is incomplete or otherwise imperfect.

$$\begin{tikzcd}
\textit{Does smoking increase blood pressure?} \arrow[r] & Y^{T=1} - Y^{T=0} \arrow[r, "\textbf{Identification}"] & \mathbb{E}[Y \mid T=1] - \mathbb{E}[Y \mid T=0] \arrow[r, "\textbf{Estimation}"] & 4.2 \text{ mmHg} \\
\text{(Causal Question)} & \text{(Causal Estimand)} & \text{(Identified Estimand)} & \text{(Causal Estimate)}
\end{tikzcd}$$

Figure 2.7: An illustration of the typical causal inference workflow, adapted from the identification-estimation flowchart presented in Neal's Introduction to Causal Inference [2].

Now that we have provided an informal overview for the typical workflow of causal inference, in the following section, we turn our attention to the foundations of this approach within the potential outcomes framework [56].

2.2.2 Causal Inference Fundamentals in the Potential Outcomes Framework

Central to causal inference is the estimation of so-called causal effects in the presence of various forms of bias. Informally, the causal effect of a particular treatment T on an outcome Y is the change in Y caused by a specific change in T. In this context, a 'treatment' is a variable that represents a particular action or intervention, such as prescribing a drug to a patient, and an 'outcome' is an observable feature or event, such as the patient's cholesterol. Causal effects can be defined for an individual, i, or a population of individuals, where an 'individual' refers to a single unit of interest, such as a person.

To provide a formal overview of the goal of causal inference, we turn to the potential outcomes framework [56]. In the potential outcomes framework, the **Individual Treatment Effect** (ITE) of a dichotomous (two-valued) treatment T is defined as the difference between the potential outcomes $Y_i^{T=1}$ and $Y_i^{T=0}$, where the potential outcome $Y_i^{T=x}$ is the outcome that would have occurred had the individual i received treatment-level x. The term treament-level is used to refer to a particular value of treatment.

Formally, we define the ITE as follows [60]:

$$\text{ITE} = Y_i^{T=1} - Y_i^{T=0} \tag{2.2.1}$$

In Equation (2.2.1), $Y_i^{T=x}$ carries a causal interpretation, namely that individual i has been hypothetically *intervened upon* by the treatment $T = x$. That is, they have been actively assigned the treatment level x. It is necessary to use this notation as the language of probability used for conventional statistics cannot express the act of doing or intervening (i.e. *physically* setting the value of some random variable) [61]. This causal nomenclature is essential to precisely defining causal actions and, critically, setting these apart from purely statistical concepts.

The term 'potential outcome' arises from the observation that, before some dichotomous treatment is administered, a given individual has two options for treatment: $T = 1$ and $T = 0$. Therefore, at this point in time, there are two separate outcomes that the individual could hypothetically observe if they were to take a particular version of treatment: $Y_i^{T=1}$ or $Y_i^{T=0}$. Of these two potential outcomes, we refer to the outcome that is actually observed as the *factual outcome* while its unobserved counterpart is known as the *counterfactual outcome*.

It follows that, in most cases, it is not possible to observe both the factual and counterfactual outcome of a dichotomous treatment for a given individual, i, $Y_i^{T=1}$ and $Y_i^{T=0}$. This is because, at any fixed point in time, the individual cannot receive both forms of treatment. Therefore, it is seldom possible to directly measure the ITE, giving rise to what is known as the *fundamental problem of causal inference* [62].

Since we cannot generally measure individual-level causal effects, causal inference usually focuses on population-level causal effects, such as the **Average Treatment Effect** (ATE). This measure of the causal effect contrasts the potential and counterfactual outcomes over some population of individuals. Formally, the ATE is defined as follows:

$$\text{ATE} = \mathbb{E}[Y^{T=1}] - \mathbb{E}[Y^{T=0}] \tag{2.2.2}$$

However, in its current form, the ATE in Equation (2.2.2) cannot be directly measured. This is again a result of the fundamental problem of causal inference [62]. Namely, that given a population of individuals, it is not possible to measure both $\mathbb{E}[Y^{T=1}]$ and $\mathbb{E}[Y^{T=0}]$ (i.e. the average outcome of the *entire* population, had they all been assigned treatment $T = 1$ and $T = 0$, respectively) as only one value of treatment can be observed per individual.

This brings us to the first task of causal inference that we briefly discussed in Section 2.2.1: *identification*. In order to estimate a causal effect of interest, such as the ATE, we need to supplement our available distribution of data with a number of assumptions, known as the *identifiability conditions*. If satisfied, these assumptions permit us to express the desired causal effect as a closed-form statistical expression that can be estimated directly, and we say that the causal effect can be *identified*. We discuss these conditions in greater detail in the following section.

2.2.3 Identifiability Conditions: Assumptions of Valid Causal Inferences

In order to estimate a causal effect, there are three so-called identifiability conditions that must be satisfied: *exchangeability*, *positivity*, and *consistency* [58]. To help define these conditions, we now outline a short example.

Example 2.1 (*Confounded Vaccination*)

Suppose an epidemiologist wants to answer the following causal question: *"Is vaccine A more effective than vaccine B at reducing the likelihood of developing critical illness C six weeks after infection with disease D?"*

For simplicity, let V be a dichotomous variable where $V = A$ and $V = B$ denote administration of two different vaccines, A and B, respectively. Furthermore, let C be a dichotomous outcome where $C = 1$ and $C = 0$ denote the outcomes of developing critical illness and not developing critical illness, respectively.

Suppose the epidemiologist has access to passively collected data that contains: (i) vaccination record ($V = A$ or $V = B$); (ii) whether or not they have developed critical illness after six weeks ($C = 0$ or $C = 1$); and (iii) a set of other variables H that may influence both the probability of receiving treatment, $P(V)$, and the probability of developing critical illness, $P(C)$.

For instance, H may include health-related factors such as age, weight, smoking status, and so forth.

Exchangeability

In order to draw valid causal inferences, it is essential that the treatment and control groups are comparable in terms of all factors that influence the assignment of treatment. Formally, this requirement is captured by the exchangeability (or unconfoundedness in the potential outcomes framework) condition. The core principle behind exchangeability is that, when it holds, had the assignment of treatment been reversed (e.g. the treatment group had received a placebo and the control group had received the active drug), the difference in outcomes would remain the same i.e. in theory, the treatment and control groups can be swapped without altering the outcome.

More formally, exchangeability requires the potential outcome $Y^{T=t}$ to be independent of treatment T [58]:

$$Y^{T=t} \perp\!\!\!\perp T \tag{2.2.3}$$

Here, it is important to emphasise that it is the *potential outcome* that should be independent of treatment, not the observed outcome (which we naturally expect to be affected by the treatment). Put differently, this means that, had the treatment assignment been swapped, the outcome would remain the same.

Exchangeability also extends to conditional settings where, only upon adjusting for the common causes of the treatment and outcome (denoted Z), are the treatment and potential outcomes independent. This version of exchangeability is often referred to as conditional exchangeability. We refer to the set of variables Z as *confounders*[3] [63]:

$$Y^{T=t} \perp\!\!\!\perp T \mid Z \tag{2.2.4}$$

In the context of our running example, exchangeability is satisfied if $C^{V=v} \perp\!\!\!\perp V$. This essentially states that, had the vaccine assignments been swapped, the outcome should remain the same.

However, we know that there is a set of health-related factors H that influence both the treatment assignment, $P(V)$, and the probability of developing critical illness, $P(C)$ i.e. H is a confounder. Therefore,

[3]We discuss the concept of confounding variables in greater detail in Section 2.2.6, where we introduce Pearl's graphical approach to causal inference.

while exchangeability does not hold ($C^{V=v} \not\perp\!\!\!\perp V$), conditional exchangeability does (provided there are no other variables that influence $P(T)$): $C^{V=v} \perp\!\!\!\perp V \mid H$. This slightly different nomenclature essentially states that, had the vaccine assignments been swapped *for groups of individuals with comparable health-related factors* H, the outcome would remain the same.

When exchangeability or conditional exchangeability is satisfied, we are able to take our causal estimand for the ATE from Equation (2.2.2) and express it as a closed form statistical expression. For example, where exchangeability holds, we can express the ATE as follows:

$$\text{ATE} = \mathbb{E}[Y \mid T = 1] - \mathbb{E}[Y \mid T = 0] \tag{2.2.5}$$

In a similar vein, where conditional exchangeability holds, we can express the ATE as follows:

$$\text{ATE} = \sum_{z \in Z} \mathbb{E}[Y \mid T = 1, Z = z]P(Z = z) - \mathbb{E}[Y \mid T = 0, Z = z]P(Z = z) \tag{2.2.6}$$

Equation (2.2.6) is one particular expression for calculating the ATE in the presence of confounding by a set of variables Z using a method known as standardisation. We discuss estimation in greater detail in Section 2.2.7.

At this point, we say that our causal estimand is identified: it can be expressed as a closed-form statistical expression that can be computed using conventional statistical techniques [64, 58]. However, although the causal estimand can, in theory, be estimated at this point, we must still satisfy two additional assumptions before we can provide a valid causal estimate: positivity and consistency.

Positivity

The second condition for valid causal inferences is *positivity* [65]. This condition states that, for a treatment T, there must be a non-zero probability of being assigned to each treatment level involved in the causal question of interest, within each stratum of the set of variables Z required for conditional exchangeability (Equation (2.2.4)) [58]. Formally, this condition is defined as follows:

$$\forall z \in Z : P(Z = z) > 0 \implies P(T = t \mid Z = z) > 0 \tag{2.2.7}$$

Positivity follows logically from the fact that, if we have only observed outcomes for one version of treatment (e.g. if all of our population have been administered vaccine A), we are missing data for other versions of treatment and therefore cannot estimate the causal effect of interest. Thus, in order to satisfy positivity, we need to observe all relevant levels of treatment within each stratum of our population, as defined by the set of variables needed for conditional exchangeability, Z.

Returning to our example, for positivity to be satisfied, we require that within each stratum defined by H (comprising the various health factors such as age, weight, smoking status etc.) we observe both versions of the vaccine: $V = A$ and $V = B$. Otherwise, for some stratum of H, we have only observed one of $V = A$ or $V = B$: we are missing the data necessary to answer our causal question. When positivity fails in a particular stratum, we can either restrict our causal inference to the stratum for which positivity holds, or extrapolate (or interpolate) from well-specified statistical models. The use of statistical models for causal inference will be discussed in Section 2.2.7.

To illustrate this point more precisely, suppose we only need to adjust for smoking status S, which is a dichotomous variable (i.e. $S = 1$ denotes a smoker and $S = 0$ a non-smoker). In this instance, positivity holds if the smoking population and non-smoking population both contain at least one individual that has received vaccine A and at least one[4] individual that has received vaccine B i.e. within each stratum of S, there is a non-zero probability of receiving both treatments involved in the causal question ($V = A$ and $V = B$):

$$P(V = A \mid S = 1) > 0,\ P(V = B \mid S = 1) > 0 \quad \text{(Positivity for smokers)} \tag{2.2.8}$$

$$P(V = A \mid S = 0) > 0,\ P(V = B \mid S = 0) > 0 \quad \text{(Positivity for non-smokers)} \tag{2.2.9}$$

Consistency

The third condition for valid causal inferences is *consistency*. Informally, this condition requires that the interventions and data are consistent with the causal question of interest [58]. In practice, this means that the outcome observed for all individuals I that receive a particular value of treatment $T = t$, are *actually* the outcomes observed by administering the hypothetical treatment t that the causal question is interested in. This condition can be formally denoted as follows:

$$\forall i \in I \text{ that receive treatment } T = t,\ Y_i^{T=t} = Y \tag{2.2.10}$$

Hernán and Robins [58] explain that consistency can be broken down into two closely related requirements: (1) the potential outcome $Y_i^{T=t}$ must be well-defined, and (2) the outcomes observed in the data should correspond to this well-defined potential outcome. The first requirement can be fulfilled through careful specification of the treatment of interest. For example, we could improve the causal question from our running vaccination example by more accurately describing the versions of treatment (i.e. vaccines A and B) and how they should be administered (e.g. the precise dose). This reduces ambiguity that could lead to different interpretations of the same causal question and, subsequently, different analyses that lead to different results.

The second requirement can be fulfilled by enforcing the conditions stipulated by the first requirement during analysis. That is, only individuals that received the precise, intended version of treatment are considered to have the treatment level $T = t$ within the data. In practice, enforcing this condition requires a well-defined data collection strategy to be defined in advance of any experiment (see Section 2.2.4) or, in the case of observational studies (see Section 2.2.5) where the data is passively collected, some additional information is required about the treatment (e.g. the dose administered).

The condition of consistency is difficult to verify in practice as what is considered to be a "well-defined" treatment is ambiguous and depends entirely on the goals of the inference. The key point of the consistency assumption, however, is to reduce ambiguity by setting a clear scope for the causal question of interest (i.e. by providing a clear description of the treatment) and, in turn, ensure the data and population is appropriate for (i.e. consistent with) this question.

Summary

The implications of the three conditions necessary for valid causal inference — namely, *exchangeability*, *positivity*, and *consistency* — can be informally summarised as follows. First, to satisfy exchangeability, we need to identify and adjust for all variables that influence both the assignment of treatment and the outcome of interest. We commonly refer to this set of variables as the adjustment set. This permits us

[4]In practice, we must also be careful of *near-violations* of positivity. That is, having a single data point for each treatment-confounder stratum would technically satisfy positivity but lead to estimation errors.

to express our causal estimand as a closed-form statistical expression. Second, once we have a sufficient adjustment set, we need to consider positivity: does our data or experimental procedure ensure there is a non-zero probability of every treatment level being observed within every stratum defined by the adjustment set? Third, provided positivity is satisfied, we must consider consistency: is the treatment sufficiently well-defined and does our data or experimental procedure match this definition?

When all three conditions are met, we can consistently estimate a causally-valid value for the desired causal estimand, such as the ATE (Equation (2.2.2)). Now that we have an understanding of the conditions that make a causal inference "causal", in the forthcoming sections, we describe two general approaches for conducting causal inference: randomised control trials and observational studies. We focus on explaining how these two approaches satisfy the identifiability conditions introduced in this section and use the same structure to facilitate a direct comparison of the two approaches.

2.2.4 Causal Inference with Randomised Experiments

Randomised Controlled Trials (RCTs) have historically been the dominant approach for conducting causal inference. An RCT is an experiment in which a group of individuals are assigned to one of two groups at random [66]. The first group, known as the treatment group, are administered the treatment, denoted $T = 1$ (such as a drug or diet). The second group, known as the control group, are administered an alternative treatment (such as a placebo drug or a different diet), sometimes referred to as a control, which is denoted by $T = 0$. After some pre-defined period of time, the treatment and control groups are compared to see whether there is any difference in the outcome of interest, Y (such as disease status or weight).

One of the main reasons RCTs are often considered the gold standard for causal inference [67] is that they satisfy exchangeability and positivity by design, enabling them to establish valid causal conclusions [68]. Furthermore, although not satisfied by design, a well-conducted RCT will also satisfy the consistency condition [68]. We now explain how RCTs satisfy these conditions and highlight one of the key limitations surrounding RCTs.

Exchangeability

Recall that for exchangeability to be satisfied, the assignment of treatment to individuals must be independent of their potential outcome. Essentially, this means that the treatment and control group should be comparable in terms of all variables that could bias the result (i.e. variables that influence both the treatment assignment and the outcome, which are formally referred to as *confounders*). RCTs achieve exchangeability through the randomised assignment of individuals to the treatment and control group. With a sufficiently large population size [69], this ensures that any potential confounders will be evenly distributed between the treatment and control group on average [67], making them 'exchangeable'.

Positivity

Positivity is the requirement that there is a non-zero probability of receiving each treatment level in the population of interest. This is trivially satisfied in RCTs as, by design, a group of individuals are assigned to each level of treatment at random. Take a dichotomous treatment, such as the vaccine from our running example: in an RCT, 50% of the individuals would be assigned vaccine A at random, and 50% vaccine B. Thus, by design, outcomes are observed for every level of treatment in the population of interest.

Consistency

A properly designed and well-conducted RCT should also satisfy the consistency assumption [68]. As a reminder, informally, consistency is the condition that the treatment is well defined and the outcomes observed correspond to outcomes of individuals who have been administered precisely this treatment. When performing an RCT, it is essential to design an appropriate research protocol, which outlines how the various stages of an RCT, from data collection to outcome measurement, should be conducted [66]. Following a well-defined research protocol helps to maintain consistency across the various steps of an RCT and therefore reduces the chance of violating the consistency assumption.

Limitations

While RCTs generally satisfy the identifiability conditions by design, there are many situations in which an RCT cannot be conducted due to practical or ethical concerns [70]. For example, many causal questions would be unethical to answer using experimental means, such as: "Does smoking cause lung cancer?". To perform an RCT capable of answering this causal question, we would have to force individuals in the treatment group to smoke and those in the control group to not smoke. Clearly, such an RCT would be unethical and would therefore not be conducted in practice.

Therefore, although RCTs are generally regarded as the gold standard for causal inference due to their ability to satisfy the identifiability conditions, there are certain causal questions that cannot be answered by this approach. This raises the question: when an RCT is not possible, how can we answer causal questions?

2.2.5 Causal Inference with Observational Data

In situations where RCTs cannot be conducted due to practical or ethical concerns, researchers often turn to observational studies to answer their causal questions. Rosenbaum defines an observational study as "an empiric investigation of effects caused by treatments when randomized experimentation is unethical or infeasible" [71]. The defining feature of an observational study is the use of observational data. That is, data in which the assignment of the treatment of interest is *passively* observed rather than the result of a physical intervention that was applied for the express purpose of the investigation.

For example, suppose we are once again interested in the causal question: "Does smoking cause cancer?". Furthermore, assume we possess some observational data collected from a health survey in which individuals volunteered to share general health information, such as smoking status, frequency, and other health factors, such as age and known health conditions. This type of data is referred to as 'observational' because the treatment value of individuals (i.e. whether they smoke or not) is passively observed: the researcher had no influence over whether or not the individuals smoke. The lack of an explicit randomised intervention in the data calls the identifiability conditions from Section 2.2.3 into question.

Practically speaking, this means that if we are to draw *causal* rather than merely *associational* conclusions from observational data, we must do additional work to justify that the identifiability conditions are satisfied. Let us now consider the additional work involved in satisfying each of the identifiability conditions within the context of an observational study.

Exchangeability

Unlike an RCT, in which exchangeability is satisfied by design through the randomised assignment of individuals to the control and treatment groups, in most situations, observational data does not naturally satisfy exchangeability. This stems from the fact that it is not generally known how or why individuals acquire a particular treatment status recorded in observational data. Indeed, there are often confounding background factors that make particular individuals more or less likely to receive a treatment that also influence the outcome of interest. These must be identified and adjusted for.

For example, socioeconomic status is known to be associated with both the prevalence of smoking [72] and occupational exposure to carcinogenics [73], which are in-turn associated with lung cancer [74, 75]. This makes the socioeconomic status of an individual a confounder, meaning that exchangeability is only achieved conditional on socioeconomic status: that is, within separate socioeconomic groups, the individuals are exchangeable, but not between groups. However, this is just one example of a potential confounder for the effect of smoking on lung cancer. There are many other potential confounders that must be considered in order to satisfy exchangeability and answer our causal question.

This example demonstrates arguably the greatest barrier to applying causal inference to observational data: *the need to identify confounding variables in order to satisfy exchangeability*. This is a process that calls upon significant domain knowledge and expert judgement, as it is only through the careful consideration of the suspected relationships between the treatment, outcome, and various covariates (variables that are associated with either the treatment, outcome, or both) that we can identify and appropriately mitigate confounding bias. To this end, in Section 2.2.6, we introduce Judea Pearl's graphical approach to causal inference, which pioneers the use of graphical causal models that assist with the identification of confounding variables based on domain expertise.

Positivity

While positivity is typically satisfied by design in RCTs, positivity violations are commonplace when dealing with observational data. There are two key reasons for this. First, due to the lack of control a researcher has over observational data, it may contain gaps or missing values that are necessary to estimate the causal effect of interest. Second, in order to satisfy conditional exchangeability, we need to adjust for all known confounders. As the number of known confounders grows, however, the observational data must be divided into smaller strata. This increases the likelihood of a positivity violation as the number of individuals per strata decreases [2]. This phenomenon is referred to as the curse of dimensionality.

To address positivity violations, a number of approaches can be taken. First and most simply, we can restrict our analysis to strata that satisify positivity. This essentially corresponds to narrowing our causal question to ask about the effect of the intervention within a particular subgroup, such as: "Does smoking cause lung cancer in individuals with lower socioeconomic status?". This approach is referred to as restriction [76]. Second, we can specify a parametric statistical model to 'fill gaps' in the data. This approach makes it possible to answer our original causal question but introduces a new assumption that must be satisfied: no model misspecification. We discuss this approach in greater detail in Section 2.2.7, where we provide a brief overview of methods for estimating causal effects.

Consistency

Due to the passively observed nature of observational data, the exact version of treatment that an individual has received may be unclear, calling the consistency assumption into question. That is, whether the version of treatment recorded in the data matches the precise version of treatment we are interested in. To address this issue, ideally, an observational study will collect additional data related to the treatment of interest that can be used to justify the consistency assumption.

For example, returning to our smoking vignette, suppose we consider an individual to be a smoker if they have been smoking at least twice a day on average for at least two years. In addition to recording whether an individual currently smokes, our observational data also contains entries for the duration of smoking (i.e. how many years an individual has been smoking for) and the approximate intensity (i.e. how many times the individual smokes per day, on average). This data can help us to identify individuals who more precisely match our definition of the treatment by excluding those that have recently taken up smoking or smoke infrequently.

The key point here is that in observational studies, although consistency is difficult to satisfy, it is best practice to define the treatment clearly and, using the available data, justify why the individuals in the observational data satisfy this definition of treatment. In some cases, it may also be appropriate to leverage the treatment-variation irrelevance assumption [77] i.e. to argue that two alternative versions of the same treatment would have the same effect and can therefore be considered as equivalent. Additionally, in the field of epidemiology, it has been recommended that potential variations of the treatment of interest are identified [78] and, whenever possible, tested [77] to confirm these assumptions.

Summary

Where causal questions cannot be answered using an RCT due to ethical or practical reasons, observational studies can be used instead. However, unlike with an RCT, the identifiability conditions discussed in Section 2.2.3 are not satisfied by design in observational studies. Thus, in order to recover causal estimates from observational data, additional effort is required to satisfy the identifiability conditions.

Most notably, it is essential to employ domain expertise in order to identify potential confounding factors to justify that conditional exchangeability is satisfied. To this end, in the following sub-section, we turn our attention to a graphical framework [59] for identification that utilises lightweight 'dot-and-arrow' models of causality whose structure can be exploited to (semi-) automatically identify confounding variables. These graphical causal models play a key role throughout the core chapters of this thesis.

2.2.6 Identification with Graphical Causal Models

As alluded to in the previous sub-section, the process of satisfying exchangeability involves identifying confounding variables. This can be an arduous process and typically requires significant domain knowledge. However, only upon identifying and adjusting for all known confounding variables can we endow our estimate with a causal interpretation [58]. Otherwise, there are uncontrolled background factors that may introduce bias to the true causal effect.

Until now, we have described the concept of confounding variables informally as variables that influence both the assignment of treatment and the outcome of interest. In this section, we introduce the notion of a causal **Directed Acyclic Graph** (DAG) - a graphical model of causality that allows us to provide a visual intuition for the concept of confounding bias. We then proceed to introduce a graphical criterion that is equivalent to exchangeabiltiy and can be used to check whether a particular set of variables, commonly referred to as an adjustment set, are sufficient to isolate the causal effect of interest (i.e. satisfy exchangeability).

This sub-section is relevant to later chapters of the thesis in which: (1) we propose the use of this graphical causal framework as the basis of a causal framework for testing computational modelling software (see Chapter 4); (2) we build upon this framework to provide a model-based approach for identifying metamorphic relations (see Chapter 5); and (3) we leverage this framework in order to apply causal inference to observational data collected from real-world computational modelling software, facilitating efficient prediction of metamorphic test outcomes (see Chapter 6).

Throughout this sub-section, we focus exclusively on the exchangeability criterion and, for simplicity, assume that positivity and consistency are satisfied. We take this decision because arguably the main advantage of causal DAGs is their ability to identify a set of variables that, once adjusted, satisfy conditional exchangeability. On the other hand, causal DAGs do not ordinarily assist in detecting positivity or consistency violations.

Causal DAGs

In the absence of an RCT, the task of discerning causation from association is a notoriously challenging task that calls upon significant domain expertise. Specifically, we need to identify and control for variables that bias the causal relationships of interest. This is particularly challenging when dealing with complex phenomena that accumulate large quantities of data involving many variables and relationships.

To address this problem, researchers in the field of causal inference developed the causal DAG [58, 59]. Causal DAGs provide a means to capture potential causal relationships between variables and are supported by an extensive mathematical framework. In particular, this framework can (semi-)automatically identify biasing variables and, thus, help to design statistical experiments capable of quantifying salient cause-effect relationships. We formally define the causal DAG in Definition 2.2 below.

Definition 2.2 (*Causal DAG*)

A causal DAG G is a directed acyclic graph $G = (\mathbf{V}, \mathbf{E})$ comprising a set of nodes representing random variables, $\mathbf{V}$, and a set of edges, $\mathbf{E}$, representing causality between these variables, where:

1. The presence/absence of an edge $V_i \rightarrow V_j$ represents the presence/absence of a direct causal effect of V_i on V_j.
2. All common causes of any pair of variables on the graph are themselves present on the graph.

As an example of a causal DAG, consider Figure 2.8. Here, $\textcircled{\texttt{X}}$, $\textcircled{\texttt{Y}}$, and $\textcircled{\texttt{Z}}$ are nodes representing *random variables*, which, in this context, are variables that can take different values for different individuals (e.g. people or software executions). We say that `X` is a *direct cause* of `Y` because there is an edge from `X` directly into `Y`. Similarly, `Z` is a direct cause of both `X` and `Y`. We refer to `Y` as a *descendant* of `Z` and `X` because there are sequences of edges, known as a *paths*, such that, if you follow the direction of those edges, you can reach `Y` from `Z` and `X`. There are three paths into `Y` in this example: $\texttt{Z} \rightarrow \texttt{X} \rightarrow \texttt{Y}$, $\texttt{X} \rightarrow \texttt{Y}$, and $\texttt{Z} \rightarrow \texttt{Y}$. And there is one path into `X`: $\texttt{Z} \rightarrow \texttt{X}$. By contrast, there are no paths into `Z`, since `Z` has no direct causes.

The utility of causal DAGs stems from an additional assumption that links its structure to the data generating process that it models. This assumption is known as the causal Markov condition, which we define in Definition 2.3 below [58].

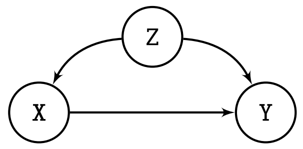


Figure 2.8: An example causal DAG for the causal effect of `X` on `Y` confounded by `Z`

Definition 2.3 (*Causal Markov Condition*)

Given a causal DAG $G = (V, E)$ and the probability distribution P over V, we say that G satisfies the causal Markov condition relative to P if, for all nodes $v \in V$, conditional on its parents, v is independent of its non-descendants in P.

Informally, the causal Markov condition means that, provided a causal DAG is an accurate representation of its subject, its structure implies a series of conditional independence relations that should hold in the data. It is this property that arms causal DAGs with the ability to identify a set of variables that, once adjusted for, satisfy the conditional exchangeability condition (Equation (2.2.4)) introduced in Section 2.2.3. To achieve this, we now introduce a pair of graphical tests: the *back-door criterion* and *d-separation*[5].

Back-Door Criterion

In order to estimate the causal effect of `X` on `Y`, we need to identify and adjust for all variables that bias the relationship `X` $\rightarrow$ `Y`. Given a causal DAG, we can identify such a set of variables by applying a pair of graphical tests, known as the *back-door criterion* and *d-separation*, which are formally defined as follows:

Definition 2.4 (*d-separation*)

A path p is *blocked* or *d-separated* by a set of variables $\mathbf{A}$ if and only if at least one of the following conditions hold [59]:

1. p contains a chain $X \rightarrow Y \rightarrow Z$ or a fork $X \leftarrow Y \rightarrow Z$ where $Y \in \mathbf{A}$.
2. p contains a collider $X \rightarrow Y \leftarrow Z$ where $Y \notin \mathbf{A}$ and for all descendants Y' of Y, $Y' \notin \mathbf{A}$.

Definition 2.5 (*Back-Door Criterion*)

A set of variables $\mathbf{A}$ is said to satisfy the *back-door criterion* relative to an ordered pair of variables (X, Y) if both of the following conditions hold [59]:

1. No variable in $\mathbf{A}$ is a descendant of X.
2. $\mathbf{A}$ blocks every path between X and Y that contains an arrow into X.

The second condition of Definition 2.5 describes what is commonly referred to as a *back-door path*.

[5]Note, d-separation is simply the name of the graphical criterion and d does not denote a parameter.

Relative to a particular cause-effect relationship $X \rightarrow Y$, we say that there is confounding if there is an open (i.e. not blocked) back-door path [58]. A confounder is then any variable that can be adjusted for to block such as path.

A set of variables Z is said to be a *sufficient adjustment set* relative to a pair of variables (X, Y) if adjusting for Z blocks all back-door paths between X and Y, without blocking the effect $X \rightarrow Y$ itself. Conceptually, this corresponds to a set of variables that, once adjusted for, mitigate all known sources of bias and that is therefore capable of isolating the *causal effect* of interest (i.e. satisfies conditional exchangeability). For example, in Figure 2.8, $\mathtt{Z}$ satisfies the back-door criterion relative to $(\mathtt{X}, \mathtt{Y})$ because $\mathtt{Z}$ blocks every path between $\mathtt{X}$ and $\mathtt{Y}$ with an arrow into $\mathtt{X}$, and $\mathtt{Z}$ is not a descendant of $\mathtt{X}$. As demonstrated in Equation (2.2.6), this permits us to express the ATE as the following closed-form statistical expression:

$$\text{ATE} = \sum_{\mathtt{z} \in \mathtt{Z}} \mathbb{E}[\mathtt{Y} \mid \mathtt{X} = \mathtt{1}, \mathtt{Z} = \mathtt{z}] P(\mathtt{Z} = \mathtt{z}) - \mathbb{E}[\mathtt{Y} \mid \mathtt{X} = \mathtt{0}, \mathtt{Z} = \mathtt{z}] P(\mathtt{Z} = \mathtt{z}) \tag{2.2.11}$$

While we focus on the back-door criterion, it is also worth noting that alternative graphical criteria exist that can be used to identify causal effects under different conditions. For example, in some situations, we may have unobserved confounders (i.e. those that we know exist, but do not have data for) that cannot be adjusted for, which prevent us from applying the back-door criterion (i.e. conditional exchangeability fails). In this situation, a criterion known as the front-door criterion [59] can sometimes be used, provided an alternative set of conditions hold.

Summary

Informally, causal DAGs are intuitive and lightweight models of causality supported by a powerful mathematical framework. In particular, the back-door criterion is a graphical test that enables us to systematically convert domain knowledge in the form of a causal DAG into statistical procedures that are capable of making *causal estimates*. This is equivalent to specifying a set of variables that are sufficient for satisfying the conditional exchangeability criterion outlined in Section 2.2.3. While causal DAGs are not essential for causal inference, they help to make key assumptions transparent and contestable in a way that the potential outcomes framework does not.

Later in this thesis, we make extensive use of causal DAGs and the back-door criterion as the basis of a model-based approach for testing computational models using causal inference. This section on graphical causal inference has presented the essential notions and definitions that are used in later chapters of this thesis (specifically, Chapters 4 to 6).

2.2.7 Estimating Causal Effects

Now that we have provided an intuition for causal inference using both the potential outcomes framework and causal DAGs, in the this section, we conclude our background on causal inference with a brief overview of different approaches to the estimation of causal effects. We focus on two broad classes of approaches to estimation: (1) estimation without the help of statistical models (i.e. with perfect and infinite data) and (2) estimation with the help of statistical models.

Estimation Without Statistical Models

Where data is complete and we have satisfied the three identifiability conditions — namely, exchangeability, positivity, and consistency — we can compute the causal effect directly using a number of approaches, including: standardisation, matching, and re-weighting [79]. Throughout this thesis, we use standardisation-based approaches to estimation.

As shown in Equation (2.2.5) and Equation (2.2.6), when the ATE has been identified, we can estimate its value using a method known as standardisation. Informally, this is a two step procedure that first computes stratum-specific estimates for each stratum of the confounding variable (this step is referred to as stratification) and then combines these estimates via a weighted average. As a reminder, this can be expressed as follows when conditional exchangeability (Equation (2.2.4)) holds under the set of confounders Z:

$$\text{ATE} = \sum_{z \in Z} \mathbb{E}[Y \mid T = 1, Z = z] P(Z = z) - \mathbb{E}[Y \mid T = 0, Z = z] P(Z = z)$$

In some instances, however, it is preferable to provide a series of separate stratum-specific estimates, which are commonly referred to as **Conditional Average Treatment Effects** (CATEs) i.e. to perform stratification without standardisation. This can be expressed as follows:

$$\text{CATE} = \mathbb{E}[Y \mid T = 1, Z] - \mathbb{E}[Y \mid T = 0, Z] \tag{2.2.12}$$

Although not a focus in this thesis, for completeness, it is also worth mentioning another commonly used causal metric. Where the ATE provides an additive measure of the causal effect, the **Risk Ratio** (RR) gives a multiplicative measure. This is captured in the following causal estimand:

$$\text{RR} = \frac{\mathbb{E}[Y^{T=1}]}{\mathbb{E}[Y^{T=0}]} \tag{2.2.13}$$

Similarly, where conditional exchangeability holds under the set of confounders Z, we can estimate the value of the above causal estimand using the following formula:

$$\text{RR} = \frac{\sum_{z \in Z} \mathbb{E}[Y \mid T = 1, Z = z] P(Z = z)}{\sum_{z \in Z} \mathbb{E}[Y \mid T = 0, Z = z] P(Z = z)} \tag{2.2.14}$$

Estimation with Parametric Statistical Models

The previous section explained how we can estimate an identified causal effect, such as the ATE, directly provided our data is complete. However, when our data is high dimensional and contains many confounders, this direct approach to estimation is often infeasible [58]. This is because increasing the number of confounders requires increasing amounts of stratification (i.e. splitting the data into increasingly small sub-sets), which reduces the effective sample size for inference and increases the chance of encountering positivity violations.

To address this problem, we can turn to parametric statistical models in order to predict unobserved outcomes that are needed to estimate the causal effect of interest. However, this requires us to introduce an additional assumption known as *no model misspecification*: the statistical model must be accurately specified if we are to consider the resulting estimates to be causal. In practice, there will likely always be some extent of model misspecification so this is an approximate assumption [58].

Throughout this thesis, we primarily use linear regression models for performing the estimation step of causal inference. The remainder of this section will describe how linear regression can be used to predict the ATE (Equation (2.2.6)). Here, we use the ‘hat’ notation to represent an estimated value. For example, $\hat{\mathbb{E}}[Y \mid T = 1, Z = z]$ represents the estimated outcome in the treated population, adjusted for the set of confounders Z.

In order to compute the ATE using linear regression, we need to predict the two potential outcomes $\mathbb{E}[Y^{T=1}]$ and $\mathbb{E}[Y^{T=0}]$, which, under conditional exchangeability, are equivalent to $\mathbb{E}[Y \mid T = 1, Z = z]$ and $\mathbb{E}[Y \mid T = 0, Z = z]$, respectively. To estimate these outcomes, we start by constructing a linear regression model which regresses the outcome Y against the treatment T while including a term for every confounder Z:

$$Y \sim \beta_0 + \beta_1 T + \beta_2 Z \tag{2.2.15}$$

We then fit this model to our data (i.e. estimate the values of the parameters β_0, β_1, and β_2) using ordinary least squares (OLS). Informally, OLS searches for the set of parameter values that minimises the sum of the squared difference between the predicted outcomes and the values of the observed variables. This yields the fitted regression model below:

$$Y = \hat{\beta_0} + \hat{\beta_1} T + \hat{\beta_2} Z \tag{2.2.16}$$

Using this fitted regression model, we now estimate the values of the treated and untreated outcomes: $\mathbb{E}[Y \mid T = 1, Z = z]$ and $\mathbb{E}[Y \mid T = 0, Z = z]$ for every combination of treatment and confounder. To achieve this, we simulate a treated and untreated population by predicting the outcome under treatment and control for every individual in the data. This requires us to substitute the values of the confounders Z for each individual (denoted Z_i) into the fitted regression model of Equation (2.2.16) and then predict the outcome with $T = 1$ and $T = 0$ separately [58]. This yields an estimated treated and untreated outcome for all n individuals in the data. We can then obtain the ATE by simply estimating and then contrasting the population means of the predicted treated and untreated outcomes:

$$\widehat{\text{ATE}} = \frac{1}{n} \sum_{i=1}^{n} \widehat{\mathbb{E}}[Y \mid T = 1, Z_i] - \widehat{\mathbb{E}}[Y \mid T = 0, Z_i] \tag{2.2.17}$$

This demonstrates the intuition behind using parametric models for estimating causal effects: we can borrow data from existing similar individuals to predict unobserved outcomes, smoothing over gaps in the data. However, the functional form of the linear regression model used here (Equation (2.2.15)) carries the highly restrictive assumption that, within strata of Z, the treatment and outcome share a linear and additive relationship. In most practical cases, the relationships being modelled will be far more intricate and inter-dependant, necessitating the use of more advanced regression modelling techniques. We give a brief overview of such techniques in the following sub-section.

2.2.8 Modelling Assumptions with Regression Modelling

As mentioned in the previous sub-section, when using a parametric model to estimate causal effects, we need to ensure that the assumption of no model misspecification approximately holds. That is, our parametric model should carry assumptions that accurately depict the phenomena or causal relationship being modelled. In this section, we discuss several commonplace regression modelling techniques that can be used to capture more intricate relationships towards satisfying the no model misspecification assumption. We focus on three key concepts that are utilised in later chapters of the thesis, namely: polynomial regression, spline regression, and interaction terms.

In the previous section, we introduced the most basic form of regression model that can yield a causal estimate for the ATE: $Y \sim \beta_0 + \beta_1 T + \beta_2 Z$. This model imposes two restrictions on the functional form of the relationship being modelled: (1) the relationship is linear meaning that, within strata of the confounder Z, the relationship between X and Y can be captured using a straight line; (2) the relationship is additive meaning that the effects of X and Z on Y do not interact with each other. However, in many cases, these assumptions will not hold. Using the misspecified regression model in such cases could lead to biased estimates that cannot be given a causal interpretation. We now explain how various regression modelling techniques and strategies can be used to specify non-linear and non-additive relationships.

Modelling Non-Linear Relationships

One approach that can be used to model non-linear relationships is *polynomial regression* [80]. This approach captures the relationship of interest by applying various non-linear transformations to the explanatory variables (those that appear on the right hand side of the regression equation), such as raising variables to an exponent (e.g. squaring or cubing) or applying a logarithmic transformation [81]. For example, if our relationship between T and Y is known to follow a U-shape, we can include a quadratic term to our regression model:

$$Y \sim \beta_0 + \beta_1 T + \beta_2 T^2 + Z \tag{2.2.18}$$

However, polynomial regression is known to have unpredictable tail behaviour (i.e. at the minimum and maximum values) [80], which makes it unsuitable for modelling certain relationships where tail behaviour is important. An alternative approach that can also model non-linear relationships in this instance is *spline regression* [82].

Spline regression is a more advanced form of regression modelling in which the data is broken up into multiple contiguous regions that are separated at particular points known as *knots*. These regions of data are then modelled using a piece-wise polynomial function of degree n, which enables the precise functional form of the regression model to change between regions, while maintaining continuity at the knots (i.e. forming a continuous function with no sudden changes in the intercept at the knots) [82]. This makes spline regression particularly adept for modelling relationships where the effect (i.e. gradient) is known to suddenly change at a number of points.

A particularly common variant of spline regression is the cubic spline, which fits a third-degree polynomial to $k-1$ regions separated by k knots, denoted $x_1, x_2, \ldots, x_k$. This can be defined as a piece-wise function $S(x)$ that specifies a separate third-degree polynomial s_r for each region $r = 1, 2, \ldots, k-1$ [83]:

$$S(x) = \begin{cases} s_1(x) & \text{if } x_1 \leq x < x_2 \\ s_2(x) & \text{if } x_2 \leq x < x_3 \\ & \vdots \\ s_{k-1}(x) & \text{if } x_{k-1} \leq x < x_k \end{cases} \tag{2.2.19}$$

where:

$$s_r(x) = \beta_{0,r} + \beta_{1,r}(x - x_r) + \beta_{2,r}(x - x_r)^2 + \beta_{3,r}(x - x_r)^3 \tag{2.2.20}$$

Modelling Non-Additive Relationships

In many situations, the magnitude of the effect of one variable T on another variable Y may be moderated by a third variable Z. That is, within different strata of Z, the expected size of the effect of T on Y varies. If this is the case, we say there is some interaction between the variables T and Z. In such cases, we typically want to provide separate estimates for the effect of T on Y within each stratum of Z i.e. predict several CATEs (see Equation (2.2.12)).

A naïve approach to this problem is to perform stratification manually: separate the data into strata of Z and provide stratum-specific estimates for the ATE, yielding a CATE for each stratum. However, this approach is limited to using the data within each stratum. In practical settings where our data is likely to be high dimensional and we may have many confounders, the effective sample size of our data within each stratum would be drastically reduced, making this approach potentially infeasible. To overcome this problem, we can alternatively specify an *interaction term* by adding an extra term in the regression model which multiplies T and Z:

$$Y \sim \beta_0 + \beta_1 T + \beta_2 Z + \beta_3 TZ \tag{2.2.21}$$

To provide some intuition behind why this works, we can take the partial derivative of the above equation with respect to T, which gives us a measure of the magnitude of the effect of T on Y. Specifically, the unit change in Y caused a unit change in T (sometimes referred to as the elasticity) [84]:

$$\frac{\partial y}{\partial t} = \beta_1 + \beta_3 Z \tag{2.2.22}$$

Now, contrast this with the same derivative applied to the simple regression equation without the interaction term (Equation (2.2.15)):

$$\frac{\partial y}{\partial t} = \beta_1 \tag{2.2.23}$$

Equations (2.2.22) and (2.2.23) demonstrate that, by including an interaction term in the revised regression model (Equation (2.2.21)), our model carries the assumption that the precise magnitude of the effect of T on Y will vary depending on Z. Consequently, this approach yields separate estimates for the ATE of T on Y within strata of Z — that is, a series of CATEs — without having to manually stratify the data first. This approach carries the advantage that it can estimate CATEs using the *entire* data set by essentially learning from 'similar' data, reducing the likelihood of estimation errors due to positivity violations.

2.2.9 Machine Learning Models for Estimating Causal Effects

Although parametric models facilitate estimation of causal effects where the data is incomplete or otherwise imperfect, they impose an additional burden on the researcher and introduce the possibility of model misspecification. One alternative to parametric modelling is the use of semi- and non-parametric statistical models, such as causal forests [85], which do not require a priori knowledge of the functional form. In other words, we can employ machine learning techniques for estimation. While machine learning does not play a significant role for estimation of causal effects in this thesis, we briefly contrast its broad advantages and disadvantages with those of parametric statistical models.

The suitability of machine learning techniques for estimating causal effects resides in its ability to learn functions from data. For example, neural networks — a popular approach to machine learning — can approximate arbitrarily complex functions from data alone [86], even when the underlying function is a high order polynomial containing many interaction terms [87]. In theory, this removes the need for a domain expert to specify the functional form of a cause-effect relationship using means such as polynomial regression, greatly reducing the potential for model misspecification in the estimation step of causal inference.

In addition, so-called double (or debiased) machine learning (DML) techniques have been developed recently that are able to obtain less biased estimates in the presence of high dimensional data, making it feasible to obtain accurate causal estimates in the presence of many confounders [88, 89]. Informally, this approach involves using machine learning models to approximate two separate functions capturing: (1) the effect of the treatment and confounders on the outcome, and (2) the effect of the confounders on the treatment. This facilitates the application of existing machine learning techniques designed for estimating so-called nuisance functions (which, in this context, are the conditional means e.g. $\mathbb{E}[Y \mid T = t, Z = z]$) in a manner that reduces the impact of the curse of dimensionality [90]. Other techniques, such as causal forests [85], have been developed with a similar motivation to mitigate of the curse of dimensionality.

However, machine learning approaches to estimation also carry a number of limitations. First, it is widely known that machine learning can be expensive and potentially require large quantities of data and/or computational resources. Thus, the use of machine learning techniques usually introduces a trade-off between cost (whether fiscal or practical) and performance. Second, machine learning models often need to be configured using various hyper-parameters and settings, such as the number of hidden layers in a neural network or the appropriate kernel function for a SVM [87]. Therefore, while machine learning reduces the burden placed on the user by removing the need to specify the precise functional form of a particular cause-effect relationship, it introduces additional steps that require a different form of domain expertise. For example, in their paper applying various machine learning techniques to propensity score estimation, Westreich et al. remark that "training of neural networks is still as much an art as a science", which hinders their widespread application in causal effect estimation.

An important point to clarify here is that the more general class of semi- and non-parametric estimation models are only of benefit to the task of estimation, not identification. Informally, this is because these approaches rely on associational measures of some description to guide the selection of variables or functional form that minimise prediction error. On the other hand, identification is a task that must be guided by *causal* information in the form of some causal model, such as a causal DAG. It is a common misconception that the former association-led type of approach can be used to select confounders (i.e. to assist identification) [91].

2.2.10 Summary

In this section, we presented an overview of the fundamental topics of causal inference concerning this thesis. We started by introducing the necessary causal nomenclature and terminology before using this to define key concepts such as individual and population-level treatment effects. Following this, in Section 2.2.3, we formally defined the identifiability conditions for causal inference and described how these are satisfied by randomised experiments and observational studies in Section 2.2.4 and Section 2.2.5, respectively. To conclude our discussion on identification, in Section 2.2.6, we introduced causal DAGs, a powerful tool for identifying confounding variables that we use as the basis for a causal form of software specification in Chapter 4.

After explaining how identification permits us to design statistical procedures capable of estimating causal effects, in Section 2.2.7, we turned our attention to methods used to actually perform estimation. In particular, we focused on methods designed around parametric statistical models, such as linear regression models and spline regression models. We use such approaches extensively in Chapter 6 as a means for estimating metamorphic test outcomes in a causally valid way. For completeness, we rounded off our overview of estimation methods with a brief discussion of machine learning techniques that can be used to estimate causal effects without requiring as much input from the user.

2.3 Causal Inference in Software Testing

Many software testing activities have a fundamentally causal objective. For example, metamorphic testing involves establishing whether some transformation to the input space affects some output(s) in a manner that can be anticipated. Given the causal nature associated with such testing activities, causal inference has recently started to emerge as a promising solution to a handful of testing problems. However, despite its promise, causal inference remains underutilised in the field of software testing. In this section, we review the few precedents that apply causal inference methods to software testing activities. While our focus in this section is predominantly on software testing applications, given the sparsity of causal inference applications in this area, we use Siebert's existing review of applications of causal inference to software engineering [92] as a guide to identify work that has relevance to the wider objectives of this thesis.

We start with an in-depth discussion of causal inference-related fault localisation research. This line of work is, to the best of our knowledge, the first testing-related problem to have been addressed using causal inference techniques and provides evidence for the impact and potential benefits of causal inference in this context. Following this, we discuss two methods that focus on *explaining* software behaviour and three applications of causal inference to specific classes of software: dataflow systems, artificial intelligence, and automotive software. We then discuss the Causal Program Dependence Analysis Framework [93] which is, to the best of our knowledge, the only other causal inference-driven testing framework, apart from the one we present in Chapter 4. Finally, we end this section with a discussion of two works that have applied causal inference methods to observational software engineering data, demonstrating the potential of causal inference to answer causal questions without performing costly or infeasible experimental approaches, and highlight several key takeaways from this body of work.

2.3.1 Applications of Causal Inference to Fault Localisation

Fault Localisation (FL) is a software engineering activity in which the developer aims to locate the root cause of faults in a program [94]. With the ever growing size and complexity of software systems, FL has established itself as a notoriously time-consuming and cumbersome task. To address this cost, statistical FL techniques have been developed that, generally speaking, estimate the *suspiciousness* of particular program elements such as statements, predicates, and methods by observing the association between particular program elements being covered in test executions and the occurrence of faults. This notion is quantified by a *suspiciousness metric*. The program elements are then ranked according to the selected suspiciousness metric and presented in descending order to the developer, enabling them to focus efforts on the most suspicious program elements first.

FL is one example of a software engineering activity that has a fundamentally causal objective - namely, to establish the root cause of a fault. From a causal perspective, the goal of statistical FL can be likened to that of causal inference with observational data discussed in Section 2.2.5 - to estimate causal effects from observational data - only, here, the causal effect in question is whether executing a particular program element, such as a statement or method, will *cause* a fault to occur. The suspiciousness metric is then a measure of the extent to which the occurrence of a fault can be attributed to coverage of a particular program element. However, despite its causal roots, suspiciousness metrics are typically calculated as a measure of *association* between a particular program element and the occurrence of a fault. As discussed in Section 2.2, without the use of causal inference techniques, measures of association may be subject to confounding bias, which, in this context, can result in misleading suspiciousness scores.

To address this limitation, a number of causal inference-led statistical FL techniques have been developed. We now discuss this line of research in chronological order.

Statement-Level FL

In their seminal paper [95], Baah et al. recognised this limitation of existing association-driven approaches to FL and proposed a causal inference-based FL technique instead. The proposed approach uses a program dependence graph as an abstraction of a causal DAG (see Definition 2.2). Leveraging the structure of this graph and the back-door criterion (see Definition 2.5), the approach is able to automatically identify *confounding statements*. Intuitively, a confounding statement is one that influences both a later statement, making it execute incorrectly, and the output of the program itself. Without knowledge of this confounding structure, the later statement appears to be the cause of the fault when it is actually the earlier statement. To mitigate this confounding, a linear regression model is then constructed for each statement s in the PUT, which takes the following form:

$$Y = \beta_0 + \beta_1 T_s + \beta_2 C_s + \epsilon$$

Here, Y is a binary outcome denoting whether a test fails, and T_s and C_s are binary indicator variables that denote whether the statement s and confounding statement (if there is one) are covered by the test suite, respectively. ϵ is an error term that is uncorrelated with T_s and C_s. This linear regression model is capable of deriving the causal effect of T_s on Y by adjusting for coverage of the confounding statement C_s. Thus, the coeffcientβ_1 can be granted causal status and, as such, captures the causal effect of covering s on the test outcome Y. That is, whether executing s *causes* the test to fail.

Through a series of experiments, the proposed approach is shown to be more effective for FL in general than existing purely statistical alternatives. This suggests that FL is indeed a causal activity and thus confounding bias is a pertinent issue worth addressing. However, since the proposed approach only considers the flow of causal information through control dependencies in a program, it can only identify confounding statements that arise as a result of a program's control flow structure. Confounding may also arise through the structure of data dependencies. Therefore, a limitation of the proposed approach is that it is only capable of identifying a subset of the possible confounding statements in a given program.

In a later study [96], Baah et al. addressed this limitation by revising the underlying causal model to consider data dependencies in addition to the control dependencies from the previous work. In addition, the revised technique uses a different estimation method, known as matching, where possible. Informally, for each program statement s, this approach adjusts for confounding by finding the most similar pair of executions e and e' where e covers s and e' does not. Here, similarity is measured using the Mahalanobis distance between dynamic dependence and control information collected during the executions. When matching is not possible because a suitably similar match cannot be found, the regression model employed in the previous paper is used instead. In an empirical study, the revised approach is shown to be more effective than the previous approach and purely statistical techniques for most programs.

However, due to the amount of data associated with statement-level execution profiles, statement-level FL is notoriously costly. This cost is exacerbated by the use of the Mahalanobis distance to compute similarity in the work by Baah et al [96], which is computationally inefficient when there are many confounders (since the dimensionality of the objects being compared increases with the number of confounders).

Extension to Method-Level FL

In order to reduce the large overhead that is typically associated with statement-level FL, Baah et al.'s approach [95, 96] was later extended by Shu et al. to handle method-level FL using similar causal inference techniques [97]. Instead of using a program dependence graph as the basis for a causal graph, in this approach, a dynamic call graph is obtained and augmented with inter-method data dependencies. Due to the use of a dynamic call graph, a novel confounder selection algorithm is developed based on the back-door criterion (see Definition 2.5). This algorithm employs a heuristic which can efficiently identify a sufficient adjustment set that satisfies the positivity condition (see Section 2.2.3), a property that was overlooked in previous approaches. Indeed, in a later study, Bai et al. showed that the accuracy of Baah et al.'s original approach [95] is reduced by positivity violations [98]. Similar to Baah et al.'s initial approach [95], Shu et al.'s approach also uses a linear regression model that adjusts for all identified confounders. In an empirical study, the approach is shown to be more effective than other method-level FL techniques, further demonstrating the suitability of causal inference techniques for the task of FL.

Taking an entirely different approach, Musco et al. propose Vautrin, a method-level FL technique that uses a call graph instead of using program execution profiles to localise faults. In short, this approach performs mutation testing (see Section 5.5.1) on methods in the PUT to identify which edges of the PUT's call graph propagate errors and which do not. This is achieved in two steps: causal graph construction and FL. In order to construct the causal graph, first, the edges are deleted from the call graph, leaving only the nodes representing methods. Second, a node is added for each test in the available test suite, with an edge to the method it tests. Third, selected methods are mutated and the failing test cases are recorded (i.e. those that kill the mutant). The shortest possible path from each failing test to the mutated method is added. This essentially prunes the call graph by removing all edges that have not demonstrated causal behaviour.

Once this approximate causal graph has been constructed, the transitive closure for each failing test node is obtained. That is, the set of methods that can be reached from the failing test node. Taking the intersection of these transitive closures yields a minimal set of methods that are reachable from all failing test cases. Implicitly, the idea here is that if there is a fault in the PUT, it is most likely to reside in the methods that are reachable from all failing tests[6]. Following this, existing FL techniques are used to compute suspiciousness scores for the reduced pool of methods. In some situations, there is no suitable intersection and the approach defaults to existing FL techniques. Hence, the approach can be alternatively viewed as a pre-processing method to reduce the cost of method-level FL.

In an empirical evaluation, Vautrin is compared to five other FL algorithms. The results of this evaluation show that Vautrin reduces the effort required to identify the faulty method by an average of 14% and makes 'perfect predictions' (the faulty method is ranked as the most suspicious) more often than its counterparts. In addition, the evaluation shows that the additional overhead imposed by the computation of the call graph and its conversion to a causal graph is minimal, amounting to less than a second of computation time.

Extension to Variable-Level FL

The approaches proposed by Baah et al. and Shu et al. rely on a relatively strong assumption. Namely, that if a statement is faulty, coverage of the statement is sufficient to bring about a fault. This assumption

[6]This assumption is arguably less plausible in realistic settings where the PUT is likely to contain multiple unrelated faults. In such a setting, it is possible that there is no intersection of the methods reachable by failing tests.

is easily violated by faults that only arise when a variable takes on a specific value, such as a fault that arises due to division by zero. For this reason, the authors highlight the need for future variable-level approaches.

To address this limitation, Bai et al. introduced a NUMFL, a value-based causal FL approach for numerical software. Value-based FL presents a significantly more difficult problem for two reasons. First, in order to conduct value-level FL, extremely detailed execution profiles must be collected that record the values of all variables involved in an execution, resulting in a significant overhead. Second, unlike the statement and method-level approaches, the treatment and confounding variables in this context are not binary. Instead, they are continuous and denote the result of a numerical expression in a particular execution. This demands the use of more advanced causal inference methods capable of estimating how the effect varies across different values of the treatment, rather than contrasting a binary treatment. The characteristic curve that models this relationship between treatment value and response is often referred to as a dose response function.

Due to these two challenges, Bai et al. employ **Generalised Propensity Scores** (GPSs) [99], a causal inference method that is capable of estimating causal effects of multi-valued treatments. Put simply, the GPS is the probability of receiving a particular value of treatment T, given the confounding variables X:

$$r(t, x) = P(T = t \mid X = x)$$

This extends the notion of a **Propensity Score** (PS) from the binary context, which is similarly defined as the probability of receiving treatment $T = 1$, given the confounding variables X (i.e. an individuals *propensity* to be treated):

$$e(x) = P(T = 1 \mid X = x)$$

In previous causal FL techniques, confounding bias was corrected by adjusting for the values of the confounders directly. However, the same result (conditional exchangeability, see Equation (2.2.4)) can be achieved by adjusting for the PS. This is because exchangeability within levels of the confounders X implies exchangeability within levels of the propensity score $e(X)$ [58]. This result also holds for GPS [99].

Following this result, in order to mitigate confounding, NUMFL constructs a linear Gaussian model for the GPS: $r(t, x) \sim N(X_e\beta, \sigma^2)$, where X_e are the outputs of earlier expressions that confound the expression e. Roughly speaking, the execution data is then binned into strata with similar estimated GPSs[7]. Within each stratum, conditional exchangeability holds and therefore we can apply standard statistical methods to each strata to obtain a causal estimate. This is achieved by fitting a separate quadratic regression model to each stratum and combining the estimates by a weighted average.

Through a series of experiments, the authors show that the proposed approach out-performs other state-of-the-art statistical FL techniques. However, it also has three key limitations. First, it is limited to numerical programs and cannot be readily applied to other classes of software. Second, it requires the user to specify two separate regression models, one of which demands significant a priori knowledge in the form of a distribution that accurately models the GPS. Third, in order to estimate the parameters of this distribution, it requires a significant amount of execution data.

With the aforementioned limitations in mind, Podgurski et al. developed an alternative value-level causal FL approach known as CounterFault [100]. This approach has two distinguishing features in particular.

[7]To be more precise, the adjustment is made according to just the parameter $X_e\beta_e$ of $r(t, x) = N(X_e\beta, \sigma^2)$. However, the mathematical details behind this slight variation of the GPS method are beyond the scope of this thesis.

First, it seeks to estimate individual-level (heterogeneous) causal effects as opposed to population-level causal effects. Put simply, this involves predicting whether the output of a test would change, had the assignment in some statement been modified: a counterfactual. The authors argue that the underlying concept of this approach is easier to comprehend. Second, it is the first causal approach to FL that uses a non-parametric estimation model (random forests [101] specifically), which has the advantage of placing less of a burden on the user. However, non-parametric models are typically less efficient and can be harder to interpret.

In addition, this work follows a novel three-step procedure to create a causal graph that more faithfully represents the causal structure of the PUT. First, it converts the PUT to a form known as **Gated Single Assignment** (GSA). Informally, this ensures that all variables are only assigned a value at a unique location (i.e. not overwritten). Second, the program dependence graph of the GSA version of the PUT is obtained. Finally, to complete the causal graph, control dependences are replaced by data dependences. This process ensures that the resulting causal DAG satisfies a condition known as the causal Markov condition (see Definition 2.3), which is not perfectly satisfied by the dependence graphs used as a causal DAG in earlier FL works.

In an empirical study, the authors applied CounterFault to several numerical programs and compared its performance to various statistical FL techniques, including NUMFL. The results show that, for most subject programs, CounterFault outperforms the alternative methods and significantly reduces the cost associated with FL. Furthermore, while the computational cost of the approach is not considered in detail, the authors note that the average run time of CounterFault (45.6s) is roughly 40% less than that of NUMFL (78.6s). However, both approaches are an order of magnitude more expensive than most non-causal alternatives.

Multi-Level FL

Previous approaches have considered FL at one level of abstraction, focusing on the effect of covering statements or methods, or changing the values of variables. Disrupting this trend, Küçük et al. proposed UniVal [102], a causal inference-driven FL technique that integrates both value-level and predicate-level FL. At a high level, this is achieved by extracting the predicates from if-else statements and assigning them as variables immediately above. This simple modification makes it possible to treat predicate-level behaviour identically to value-level behaviour. Following this modification, an approach similar to that of CounterFault is employed. First, a GSA causal graph is obtained, before a non-parametric statistical model is used to estimate individual-level causal effects. However, in addition to the predicate-level functionality, UniVal also provides support for expressions involving strings.

In a large scale evaluation, the authors apply UniVal to a large number of subject programs from Defects4J [17] and compare its performance to several other statistical FL techniques, including NUMFL. However, despite being conceptually the most similar alternative, CounterFault does not feature in the evaluation. The results demonstrate that, across all programs, UniVal generally provides the most accurate ranking of statements and thus reduces the cost associated with FL to the greatest extent. Moreover, in contrast to the other approaches, there is little variance in UniVal's ranking of suspiciousness, suggesting it can locate faults consistently as well as accurately.

Summary

In the decade that followed the seminal paper of Baah et al. [95], a series of causal inference-led FL techniques have been developed, leading to a plethora of tools that increasingly outperform purely statistical alternatives. From this work, we highlight the following key developments:

- *Program dependence graphs as a model of causality*. Throughout this line of work, we have seen various forms of program dependence graphs used as causal models, starting with incomplete abstractions based on control flow and data dependence and ultimately converging on GSA program dependence graphs.
- *Levels of abstraction*. In this line of work, we have seen FL approaches that target faults at different levels of program abstraction, including at the statement, method, and value levels. We have also seen approaches that target mutliple levels of abstraction.
- *Causal estimation methods*. In the earlier causal FL works, parametric estimation methods, such as linear regression, were readily used to estimate causal effects. We also saw more advanced parametric methods used to handle more difficult situations, such as GPSs. In recent works, this trend has changed and we have observed an increasing reliance on non-parametric estimation methods, such as random forests.

There are several open opportunities in this line of work. First, only a single technique considers multiple levels of abstraction and, even so, this technique only considers two levels of abstraction. Future work could develop techniques that cover all levels of abstraction to provide a single, complete approach to FL. Second, most of the approaches are only evaluated on numerical programs. On both fronts, UniVal [102] appears to be the most mature causal FL technique, supporting value and predicate-level FL and also providing support for string-type variables. However, its performance has not been evaluated against faults involving strings.

2.3.2 Applications of Causal Inference for Explaining Software Behaviour

Once a fault has been localised, a remaining challenge is to explain *why* it occurred. This is an example of a causal question that arises in software testing.

With the aim of helping developers to *explain* the root cause of faults, Johnson et al. [103] introduced the notion of *Causal Testing*. The underlying concept of Causal Testing is to take a failing test case and find a minimal change to its inputs that causes the test outcome to change (i.e. fixes the failing test case). The corresponding minimally different but passing test case is known as a counterexample and is presented to the user as a compact explanation for the cause of failure.

More specifically, Causal Testing involves constructing so-called *causal experiments* in which the inputs of a failing test are manipulated in numerous ways to create several alternative, minimally different versions of the test case. These new test cases are executed against the same test oracle and those that now pass are collected. Of these passing test cases, only the most similar (in terms of inputs and execution path) are returned to the user alongside the resulting execution trace. The intuition here is that if the test outcome has changed and the passing test is similar to the failing one, the differences between their inputs and execution paths should pinpoint and explain the cause of the fault.

Johnson et al. also provide *Holmes*, a preliminary version of a Causal Testing tool. In their empirical evaluation, the authors found that Holmes was applicable to 71% of defects in the Defects4J framework and, of those, was capable of explaining 77% of the root causes. In addition, through a controlled experiment, the authors found that Holmes helped developers to explain 6% more faults than could be explained without the assistance of the tool.

In a similar vein, Dubslaff et al. [104] proposed a theoretical framework and developed an accompanying tool, FeatCause, for detecting and explaining[8] causal relationships at the feature level in configurable software systems. Conceptually, this involves determining which features (Boolean input conditions of the PUT) *cause* a particular observable event in the output space, such as the PUT crashing. Due to the problem of combinatorial explosion, this class of software presents a significant challenge when it comes to detecting and explaining the possible causes of functional and non-functional events.

To address this challenge, the authors draw on results from propositional logic and circuit optimisation to develop a series of algorithms that facilitate the identification of causes in configurable systems. Unlike Causal Testing [103], these algorithms do not rely on access to the source code. Instead, it requires the user to specify a set of valid feature configurations and corresponding anticipated effects on the output space as Boolean expressions. FeatCause then uses a circuit optimisation tool to compute a set of candidate causes: feature configurations that cause the expected output where, if any constituent feature is changed, the expected output will no longer occur. In addition, relative to a particular effect, FeatCause can estimate the degree of blame for each feature, roughly quantifying the magnitude of causality.

In a series of experiments, FeatCause is applied to a wide range of real-world configurable software systems and evaluated in terms of its ability to identify and explain root causes of specified effects. The results show that the approach is able to compute relatively concise explanations in a reasonable amount of time (seconds for most systems, hours for some). Moreover, the degree of blame is found to be a useful metric for assessing the relative causal impact of features and interactions amongst them.

2.3.3 Application of Causal Inference to Specific Software Systems

Aside from the line of work on causal FL, most other testing-related research involving causal inference has focused on demonstrating its suitability for appraising the quality or reliability of specific classes of systems. We now provide a brief overview of this work.

Dataflow Systems

In the 1970s, dataflow programming emerged as a new paradigm [106] in which programs are modelled as a form of directed graph known as a dataflow graph. In a dataflow graph, nodes represent operations and edges represent the flow of data through operations, where the incoming edges to an operation represent its inputs and the outgoing edges its outputs. In recent years, with the rise of big data, dataflow programming has grown in popularity as a solution for managing data in complex, unwieldy systems. For example, it has been used to orchestrate the flow of data through large-scale distributed machine learning systems [107].

Recently, Paleyes et al. highlighted the natural correspondence between dataflow graphs and causal DAGs (see Definition 2.2). In particular, the authors show how a particular variant of dataflow programming, known as flow-based programming, naturally produces a dataflow graph that can be interpreted as a complete DAG without further modification [108]. This a noteworthy observation as it removes one of the main barriers to applying (graphical) causal inference to software systems: the need to construct a

[8]To be more precise, the proposed approach provides explanations in the form of causal explications [105].

causal DAG. Therefore, dataflow programming inherently facilitates the application of causal inference techniques.

In a series of preliminary experiments, Paleyes et al. demonstrate how, using the dataflow graph of a program, causal inference can help to solve two problems in a toy commercial system: (1) FL and (2) explaining the cause of unexpected changes in key performance indicators (KPI). To achieve this, the authors propose an algorithm which iterates over the dataflow graph and calculates the attribution score for each node. Attribution can be approximated using several metrics, including the well-known Shapley value; a concept originating in the field of game theory that is commonly used in machine learning to quantify the relevant importance of each feature to a prediction [109] . Intuitively, this approach quantifies the probability that a particular node (operation) is the cause of some observed change in the output distribution (e.g. caused by the presence of a fault or a shift in a KPI). In a later paper, Paleyes et al. performed further experiments to demonstrate the validity of this approach for FL in dataflow systems [110].

Artificial Intelligence Systems

Another domain that has shown interest in the use of causal inference for testing-related activities is **Artificial Intelligence** (AI). Motivated by their own experience in applying causal inference to an industrial-academic research project, in an extended abstract, Heyn and Knauss [111] advocate the use of structural causal models as boundary objects for developing AI systems. In this context, a boundary object is some artefact that facilitates knowledge exchange between distinct users of a system, such as domain experts, engineers, and testers.

As examples, Heyn and Knauss describe the use of causal inference for two tasks involved in developing a pedestrian detection system: data acquisition and runtime monitoring. For the former task, the authors draw a causal DAG and use this to identify sources of confounding bias in training data. In particular, they find that weather is a confounder as it influences both the likelihood of a pedestrian being outside and quality of the image. This knowledge is then used during data collection to ensure that training data is collected in a manner that mitigates confounding bias (e.g. collecting additional bad weather samples).

For the latter task, the authors explain how, conceptually, the dependence relations in the modelled causal DAG can be tested using measures of statistical dependence to validate the causal assumptions. This is motivated by Pearl's [112] observation that dependencies in data can be used as evidence for the relationships modelled by a structural equation model (a variation of a causal DAG in which the functional form of every causal relationship is also specified). In Chapter 5, we employ a similar philosophy to develop a method for deriving metamorphic relations from a causal DAG representing the expected behaviour of a program.

Automotive Software

In the automotive industry, vehicles are becoming increasingly dependent on software to handle a range of tasks, from managing the deployment of battery power in hybrid vehicles to collision avoidance in autonomous vehicles. Given its safety critical nature, it is imperative that such software is adequately tested. However, it is difficult to accurately simulate the wide range of conditions a vehicle is likely to encounter within a testing environment. For this reason, to establish the impact of new or updated software in an automotive context, online randomised experiments are ideally conducted. This could take the form of an A/B experiment, where two versions of the software are randomly assigned to different users, thereby achieving exchangeability (Equation (2.2.3)) and revealing the causal effect of the software changes. However, this is a costly and often infeasible approach [113].

To address this limitation, Liu et al. [113] developed a framework for applying causal inference methods to evaluate the impact of online automotive software changes. In collaboration with an international automotive manufacturer, this work focuses on applying three advanced Bayesian methods to estimate the causal effect of software changes of a hybrid car in three different but equally challenging scenarios.

The first scenario involves evaluating the impact of software designed to increase fuel efficiency, where, for reasons of practicality, only a small number of vehicles are assigned the update. Here, the assignment of treatment (the software update) is not random and, therefore, the effect of interest may be confounded. To mitigate this bias, the authors use Bayesian PS matching (similar to in Bai et al.'s method-level FL technique, NUMFL [114]) to eliminate confounding on a number of selected variables, such as trip distance and trip frequency. This yields a negative causal estimate that indicates the software does indeed reduce fuel consumption on average.

The second scenario is centred around software designed to manage the temperature of the battery in order to ensure it operates within its optimal performance window. This scenario is complicated by (physically) uncontrollable external factors, such as weather and individual driving habits, that strongly influence (and potentially confound) the effect of interest *over time*. To address this, the authors employ a methodology known as difference-in-differences [115][9], which is suitable for estimating causal effects in longitudinal data. This yields a negative effect estimate, suggesting that the software reduces average energy consumption as expected.

The third and final scenario concerns software developed to optimise the deployment of electrical and chemical energy on a given trip. In this scenario, the performance of the software is highly dependent on the distance of the trip. For example, when the distance is long, the driver is more likely to use motorways, where the internal combustion engine is more efficient than relying on battery power. Conversely, when the distance is less than the battery range, it's generally more efficient to operate under battery power. With this in mind, the authors use a method known as regression discontinuity design[10] [116] to estimate the effect of the software on fuel consumption when deployed for trips exceeding the expected battery range. This reveals that the software reduces the average fuel consumption when compared to a naïve strategy that exhausts the battery before switching to the internal combustion engine.

This paper demonstrates the utility of causal inference methods for testing the performance of automotive software in situations where the ideal online A/B experiments are not pragmatic. Instead, the authors show how advanced Bayesian estimation methods can be used to emulate the ideal experiment. However, unlike the causal inference methods used throughout this thesis, Liu et al. focus on the use of Bayesian methods for causal inference and, most notably, leave the identification process implicit; causal DAGs are not presented as a justification for the choice of adjustment set when configuring the various estimation models.

[9]The exact details of this methodology are beyond the scope of this thesis. However, the key point to note is that it is commonly used in econometrics to estimate causal effects in longitudinal data where pre-treatment data is available [115].

[10]It suffices to know that regression discontinuity design is a method used to estimate treatment effects in situations where the assignment of treatment is known to vary about some critical threshold of a predictor variable. In this work, the predictor is the battery range and the threshold is its maximum range.

2.3.4 Causal Program Dependence Analysis

To the best of our knowledge, other than the Causal Testing Framework that we introduce in Chapter 4 of this thesis, there is only one other work that proposes a general purpose approach for testing-related activities that uses causal inference: Lee et al.'s **Causal Program Dependence Analysis** (CPDA) framework [93]. CPDA is a dynamic dependence analysis technique that strategically manipulates and then executes a program with the aim of collecting interventional data that can be used to approximate the causal structure of the PUT. This involves three main steps: instrumentation, causal structure discovery, and causal dependence modelling.

Instrumentation involves applying mutations to the program, overwriting the value a particular variable takes, and observing the impact this has on the program output. The resulting data is then used to predict the causal structure of the program, modelling each variable as a node, and observing which nodes changed in response to a particular intervention. Following this, the causal structure is refined by iteratively checking conditional independence relations to obtain a causal graph that satisfies the causal Markov condition (see Definition 2.3). Finally, using the interventional data, the strength of causal dependence is estimated using 'direct dependence', an aggregate measure of the effects observed within the interventional data. These dependence measures are then attached as weights to the edges of the causal structure, forming a **Causal Program Dependence Model** (CPDM): a weighted directed graph that, roughly speaking, quantifies the extent to which program elements cause one another.

As an example application of this framework, the authors conduct FL, proposing the notion of causal dependence-based FL. Causal dependence-based FL works on the assumption that a faulty program element is likely to have a greater causal effect on failing executions than passing executions. Given this assumption, causal dependence-based FL calculates the suspiciousness score of a program element X_1 on a fault-containing program element X_2 by calculating the average causal dependence of X_2 on X_1 twice using two separate sets of data: once with passing test executions and once with failing test executions. The two respective outcomes are then contrasted to obtain the average difference in causal dependence, which is used as the (causal) suspiciousness metric.

In an empirical evaluation, the authors show that causal dependence-based FL generally outperforms slicing-based approaches to FL and, with enough interventions (mutations) per program element, spectrum-based approaches too. In addition, to qualitatively evaluate the effectiveness of CPDA, the authors apply the approach to the widely used word count (wc) program and compare the resulting CPDM to its ordinary program dependence graph. This program is selected as it has a well understood and relatively straight forward dependence structure. The discussion that follows indicates that the dependence strengths provide a useful guide for developers looking to focus on key features, leading to improved program comprehension.

In a later work, Oh et al. showed how CPDA can be used to identify **Higher Order Mutants** (HOMs) [117]. As discussed in Section 5.5.1, a key challenge in mutation testing is the detection of equivalent mutants - a problem that is theoretically undecidable. One solution to this challenge is to perform multiple mutations per mutant, yielding a HOM. In this work, Oh et al. develop a number of heuristics based on the CPDM that can effectively guide the generation of a class of HOMs known as **Strongly Subsuming HOMs** (SSHOMs): HOMs that are more difficult to kill than its constituent **First Order Mutants** (FOMs).

The intuition behind the approach is that HOMs that target pairs of program elements that have a high causal dependence (i.e. connected by an edge in the CPDM with a high weight) are more likely to exhibit a masking effect. This is where the constituent FOMs are individually killed by the test suite, but the combination of them (HOM) manipulates the program in a way that causes the HOM to survive. Hence, the resulting HOM is harder to kill than the individual FOMs. With this in mind, the authors develop three heuristics that aim to detect SSHOMs. By applying these heuristics to CPDMs of two simple programs, the authors find that all three are more more effective in detecting SSHOMs than selecting program elements at random. More generally, this work suggests that causal effects are a useful diagnostic of program elements that can be mutated to produce SSHOMs.

2.3.5 Applications of Causal Inference to Software Engineering Data

Although not strictly related to the topic of software testing, it is worth briefly discussing two works that use causal inference methods to answer causal questions using observational software engineering data. The discussion that follows shows how causal inference methods can be used to infer causal effects from observational data *a posteriori*, enabling researchers to answer causal questions retroactively and without needing to conduct a randomised experiment.

Does Sharing an Open Source Project on Twitter Affect Popularity?

One work that applies causal inference to observational software engineering data, conducted by Fang et al. [118], uses causal inference techniques to estimate the impact of tweets on the popularity of open source projects. This work uses a particular causal inference method known as difference-in-differences estimation (as used by Liu et al. [113] in the context of automotive software) which is suitable for analysing causal effects in longitudinal data. Using this approach, the authors find that tweeting about an open source project has a statistically significant causal effect on both the number of stars and new contributors. However, the magnitude of the effect on the number of stars is significantly greater.

Does Choice of Programming Language in a Programming Competition Affect Performance?

More recently, Furia et al. [119] explored the suitability of causal inference techniques that employ structural causal models (i.e. use a causal DAG) and demonstrate the application of this technique to a well-studied problem in the field of software engineering: the impact of programming language on a participant's performance in a coding contest. This causal analysis reveals that the choice of programming language has a small but significant causal effect. Interestingly, when compared to a broadly equivalent statistical analysis, the two analyses yield near opposite conclusions for the most advantageous programming language. Regardless, the magnitude of this effect is diminished by the magnitude of other variables' effects, such as the programmer's identity, suggesting that the choice of programming language is largely irrelevant.

The main takeaway from this paper, however, does not concern the 'right' choice of programming language for a programming contest. Rather, it is the general observation that causal inference techniques and, more generally, the process of applying them, encourage us to analyse data from a different perspective that can lead to higher quality research that produces more sound conclusions. To this end, the authors advocate the use of causal inference methods for analysing the wealth of observational software engineering data, citing its ability to establish salient causal relationships over mere associations in data, in the pursuit of answers to causal questions.

2.3.6 Key Takeaways

While causal inference is not yet an established methodology in the field of software testing, in this section, we have reviewed the handful of precedents that do exist. In addition, we have reviewed relevant applications of causal inference in the wider domain of software engineering, for activities such as FL. From the common themes established in this relatively small body of research, we now answer three questions that summarise the main high-level takeaways from this work.

What are the hallmarks of a problem suitable for causal inference?
Throughout the literature reviewed in this section, there are several similarities amongst the problems that causal inference has been applied to, which may indicate the types of problem for which causal inference is most likely suitable.

First, all of the reviewed literature has focused on a problem with a *fundamentally causal objective*. We use this term throughout this thesis to describe an activity which ultimately boils down to seeking an answer to a *causal question*. For example, in the line of causal FL work, the underlying causal question is whether covering some program element *causes* a failure to occur. To answer such causal questions, it is necessary to evaluate the *causal effect* of some *change*. For example, in the automotive context, Liu et al. [113] used Bayesian causal inference methods to estimate the impact of software updates on various vehicular performance metrics, such as fuel consumption. Consequently, all of these problems share the same key ingredients: an intervention and an outcome.

Second, generally speaking, there are two distinct groups of causal inference methods that are suitable in different settings. On the one hand, there is the experimental line of research, where causal effects are measured by some form of (at least) quasi-experiment. For example, Johnson et al.'s Holmes tool [103], which generates and executes tests to find a counterexample that explains the cause of a test failure. In the context of software systems, this form of causal inference requires *the ability to execute the PUT* to isolate the causal effect of interest. On the other hand, there is the observational line of research, where causal effects are estimated from observational data. For example, Furia et al. estimated the causal effect of programming language choice on performance in programming competitions from historical data [119]. This, of course, requires access to observational data, as well the necessary domain expertise to identify and eliminate confounding variables (e.g. by drawing a DAG and performing identification; see Section 2.2.6).

Third, where observational data is used for making causal inferences, it is necessary to possess some form of causal model. The vast majority of the existing literature has relied on abstractions of a causal model that can be obtained automatically or semi-automatically. For example, in the line of causal FL work (discussed in Section 2.3.1), various forms of automatically derivable program dependence graphs were used as causal models. Conversely, there are only a couple of precedents for using hand-crafted causal models: Furia et al. [119] manually specified causal DAGs to identify confounders in software engineering data and Liu et al. [113] identified suspected confounders with the help of domain experts (however, few details of this process are given).

Therefore, in short, causal inference is suitable for software testing activities that:

1. Concern the causal effect of some intervention on some output.
2. Provide access to either:
    (a) The PUT so that it can be executed in some form of quasi-experiment to isolate the causal effect of interest.

   (b) Observational data collected from the PUT or artefacts of it so that causal effects can be inferred retrospectively.

3. Can be expressed using some model of causality, such as a causal DAG.

What advantage does causal inference provide?
It is widely known that the primary advantage of causal inference is its ability to establish causal conclusions. This corresponds to identifying and eliminating confounding bias in order to home in on salient cause-effect relationships. In this section, we have seen this advantage realised in two distinct ways.

First, causal inference has been used in the conventional sense to reduce bias in causal estimates. For example, in the context of FL where confounding bias has been a longstanding problem in the estimation of suspiciousness scores, the line of causal FL approaches have used causal inference methods to effectively mitigate confounding. Consequently, causal approaches to FL are generally more accurate and thus effective at locating faults than purely statistical alternatives.

Second, we have seen examples of causal inference being used to answer causal questions which are either impossible or infeasible to answer using experimental means. For example, while theoretically superior, Liu et al. [113] could not conduct online A/B tests to evaluate software updates deployed to a fleet of hybrid road vehicles. Instead, they used Bayesian causal inference methods to essentially emulate these tests.

Therefore, causal inference has two main benefits. First, experimental and observational approaches can mitigate confounding bias where this is an established problem, such as in FL. Second, observational approaches can be used to emulate experimental procedures in order to answer causal questions when it is not feasible to carry out the ideal experimental procedure.

Where are there opportunities to apply causal inference in software testing?
Aside from the work conducted in this thesis, there are only a handful of approaches that utilise causal inference for software testing activities. Most notably, Lee et al.'s Causal Program Dependence Analysis [93], Liu et al.'s application of Bayesian causal inference methods to automotive software [113], and Johnson et al.'s Holmes tool for explaining the root cause of test failures [103]. Therefore, the intersection between causal inference and software testing is in its infancy and there is great untapped potential.

Moreover, drawing on our answers to the previous two questions, we can begin to identify software testing activities that are suitable to translate to problems of causal inference and would benefit from this causal treatment. For example, regression testing, which establishes whether some software change has caused any previously passing tests to fail, has a clear causal objective and involves the previously mentioned key ingredients: an intervention (the software change) and an outcome of interest (new test failures). Similarly, metamorphic testing, where the goal is to establish whether transforming the input space in a purposeful way causes an expected change in the output space, shares a natural symmetry with causal inference. In Chapter 4, we explore the causal nature of metamorphic testing in greater detail and provide a more robust argument as to why it would benefit from being framed and solved as a problem of causal inference.

2.4 Concluding Remarks

In this chapter, we have reviewed literature concerning the two main topics of this thesis - software testing and causal inference - as well as the small body of research that explores their overlap.

More specifically, in Section 2.1, we outlined the fundamentals of software testing, from the components of an archetypal testing framework to measures of test adequacy. We also introduced the test oracle problem - a common testing challenge that acts as a key motivating factor in this thesis - and metamorphic testing, which is one of the few techniques designed to mitigate this problem. In addition, in Section 2.2 we introduced causal inference: a family of statistical methodologies designed to establish and measure salient causal relationships. Causal inference plays a central role throughout this thesis as we aim to understand how it can help to improve the applicability of metamorphic testing to computational modelling software.

In the forthcoming Part I, we turn our attention to formulating our problem statement and proposed solution. To this end, in Chapter 3, we study two real-world computational models to demonstrate the practical challenges that currently hinder the applicability of metamorphic testing to this class of software. Using this as evidence, we then outline our problem statement for this thesis. With this in mind, in Chapter 4, we outline the conceptual overlap between metamorphic testing and existing experimental designs for causal inference, motivating our proposed solution: to approach metamorphic testing as a problem of causal inference. In order to achieve this, we build on the fundamentals explored in this background and extend the software testing framework of Staats et al. [1] introduced in Section 2.1.1 to incorporate the elements of causality needed to apply methods of causal inference to causality-focused software testing problems, such as metamorphic testing.

Part I

Problem Statement and Proposed Solution

3 | Metamorphic Testing of Computational Modelling Software

In this chapter, we introduce two forms of computational model of differing complexity that are commonly used to model epidemiological phenomena, such as the recent COVID-19 pandemic, as motivating examples. For each of the models, we begin with an in-depth discussion of its prominent features and characteristics, and highlight the problems these pose to conventional testing methods; most notably, the oracle problem and long execution times. We then apply a form of testing that is a tried and tested solution to the oracle problem: metamorphic testing (see Section 2.1.5).

We select these models as they are a timely example of how computational modelling software is used in practice. Furthermore, previous research has established a series of metamorphic relations designed to capture the expected behaviour of epidemiological modelling software [5] that can be used as common ground to demonstrate how metamorphic testing adapts to the differing characteristics of our two example models. In doing so, we demonstrate how the intricacies of the two models lead to different testing challenges, and highlight how even the state-of-the-art testing technique struggles to adapt to these challenges.

After analysing the challenging characteristics of such models and the problems they pose in a software testing context, we conclude this chapter by outlining our problem statement.

3.1 Motivating Example I: Epidemiological Equation-Based Model

Over time, a number of pandemics and epidemics have emerged that have led to countless deaths, from the Black Death (1346 – 1353) to Spanish Influenza (1918 – 1920). More recently, the ongoing COVID-19 pandemic serves as a reminder that such events remain a significant threat, both in terms of loss of life and their potential fiscal impact, and of the need for preventative and preemptive measures that can effectively contain outbreaks. One technique that is commonly used to remain one step ahead is computational modelling of pandemic scenarios.

As an example computational model employed in this context, we now introduce a relatively simple epidemiological computational model that simulates the progression of Spanish Influenza in the United States in 1918. This model is a type of model known as an **Equation-Based Model** (EBM), where the behaviour is governed by a series of **Ordinary Differential Equations** (ODEs). EBMs are particularly suited to simulating macro-level outbreak dynamics [120]. However, they are not suitable for modelling intricate micro-level details as individuals are treated as homogeneous entities in EBMs [121].

To demonstrate how metamorphic testing can be employed within this context, we have implemented the Spanish Influenza EBM introduced by Sukumar and Nutaro in their 2012 paper entitled: "Agent-Based vs. Equation-Based Epidemiological Models: A Model Selection Case Study" [3]. We have selected this model as it is the subject of a later paper which defines a series of epidemiological metamorphic relations that can be used as a form of test oracle [5]. In Section 3.3, we test our implementation of the Spanish Influenza EBM against these metamorphic relations. Our implementation of this model can be found in our open source repository[1].

3.1.1 Model Parameters

The Spanish Influenza model is an EBM [122] comprised of a series of ODEs that characterise the progression of the outbreak over time in terms of several state variables. An ODE is a differential equation that is a function of a single independent variable [123]. In this model, ODEs are defined for the six variables described in Table 3.1.

Variable	Symbol	Description
Susceptible	S	The number of individuals susceptible to infection.
Incubating	E	The number of infected individuals that are not yet infectious.
Infectious	I	The number of individuals that have been infected and are infectious.
Recovered	R	The number of infected individuals that have subsequently recovered.
Deceased	D	The number of infected individuals that have subsequently died.
Infected	T_I	The cumulative number of infected individuals.

Table 3.1: A description of the state variables that are modelled in the Spanish Influenza EBM.

The ODEs for the state variables listed in Table 3.1 are defined in terms of the parameters shown in Table 3.2. The values for these parameters are the same as in the original study [3], in which the authors derived the initial values from historical data documenting the Spanish Influenza pandemic in the United States in 1918 [124, 125]. In this way, the parameters are considered to be the inputs of the model, while the various time series describing the flow of individuals through disease states over time are considered as the outputs.

Collectively, these parameters are used to form the system of equations shown in Equations (3.1.1) to (3.1.6). This system of equations characterise the relationship between the parameters and transitions between the states (i.e. disease progression), where $\beta = \tau\mu$ as per Table 3.2. We now informally describe the behaviour of each ODE.

$$\frac{dS(t)}{dt} = -\beta \frac{S(t)}{N} I(t) \tag{3.1.1}$$

Equation (3.1.1) computes the change in the number of susceptible individuals by calculating the number of new infections transmitted from infectious individuals (I) to susceptible individuals (S). This calculation accounts for the transmissibility of the disease (β), which is itself determined by the number of contacts (μ) and transmission probability (τ).

[1] https://github.com/AndrewC19/phd_thesis_code_examples/tree/main/models/ebm

Variable	Symbol	Description	Initial value
Incubation time	σ	Time before an incubating individual becomes infectious.	3
Death probability	α	The probability that an infected individual will be killed by the disease.	0.01
Death time	γ_D	The time for death to occur for individuals killed by the disease.	1
Recovery time	γ_R	The time for recovery for individuals who survive the disease.	2.5
Transmission probability	τ	The probability that a susceptible individual is infected upon contact with an infectious individual.	0.15
Mean contacts	μ	The average number of daily contacts per individual.	4
Population size	N	The total number of living individuals in the model at initialisation.	103267000
Initial infectious	I_0	The number of infectious individuals at the point of initialisation.	1000
Expected infections	$\beta = \tau\mu$	The expected number of infections transmitted by an infectious individual per time-step.	0.6

Table 3.2: A description of the parameters of the Spanish Influenza EBM.

$$\frac{dE(t)}{dt} = \beta \frac{S(t)}{N} I(t) - \frac{E(t)}{\sigma} \tag{3.1.2}$$

Equation (3.1.2) computes the change in the number of incubating individuals (E) by calculating the number of individuals that have been incubating for the full incubation time (σ) and thus become infectious. This pool of newly infectious individuals is subtracted from the pool of existing and newly incubating individuals.

$$\frac{dI(t)}{dt} = \frac{E(t)}{\sigma} - \left(\frac{\alpha}{\gamma_D} + \frac{(1-\alpha)}{\gamma_R}\right) I(t) \tag{3.1.3}$$

Equation (3.1.3) updates the number of infectious individuals by calculating the number of previously infectious individuals that have since died (determined by death probability α and death time γ_D) or recovered (determined by recovery probability $1-\alpha$ and recovery time γ_R). These individuals are then removed from the infectious compartment.

$$\frac{dD(t)}{dt} = \left(\frac{\alpha}{\gamma_D} + \frac{(1-\alpha)}{\gamma_R}\right) I(t) \tag{3.1.4}$$

$$\frac{dR(t)}{dt} = \left(\frac{\alpha}{\gamma_D} + \frac{(1-\alpha)}{\gamma_R}\right) I(t) \tag{3.1.5}$$

Equation (3.1.4) and Equation (3.1.5) are updated by calculating the number of infectious individuals (I) that have since died or recovered, respectively, accounting for the probability and time of death/recovery. Individuals that have died or recovered are subsequently removed from the model. This means that re-infection is not possible.

$$\frac{dT_I(t)}{dt} = \frac{E(t)}{\sigma} \tag{3.1.6}$$

Finally, the change in Equation (3.1.6) is calculated by computing the number of incubating individuals (E) that have become infectious, meaning the full duration of their incubation period has elapsed. This is not an infectious disease state, per se, but a cumulative total of the number of infected individuals over time.

3.1.2 Model Outputs

Solving the system of ODEs over 200 time steps (days) produces the time-series shown in Figure 3.1. Visually, the trends shown in Figure 3.1 appear to be in agreement with the same results presented by Sukumar and Nataro [3], which are shown in Figure 3.2 (the ODE results are shown by the red lines).

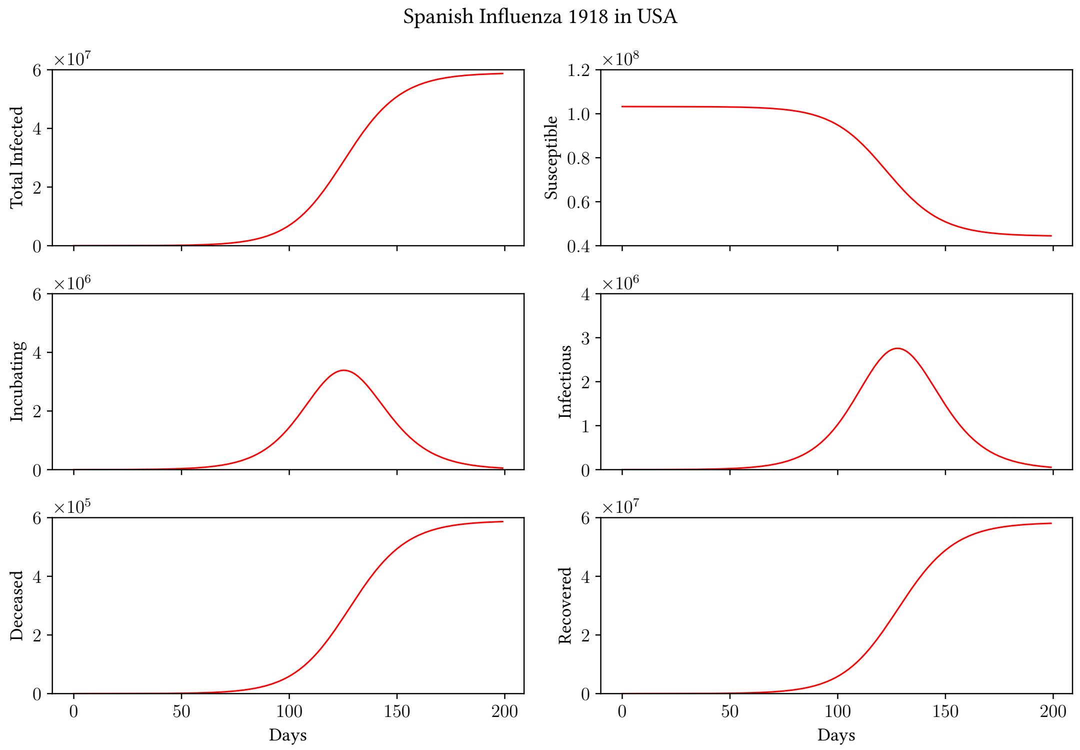


Figure 3.1: Time series produced by solving the Spanish Influenza EBM over 200 time steps using the initial conditions outlined in Table 3.2.

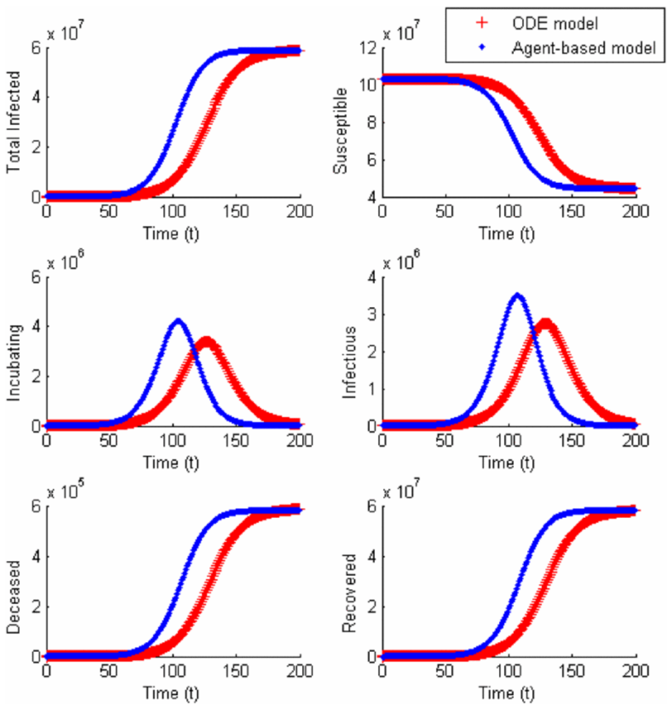


Figure 3.2: Original results for the Spanish Influenza model presented by Sukumar and Nataro [3]. © 2012 IEEE.

Now that we have described the basic mechanistic behaviour behind the 1918 Spanish Influenza EBM, including its inputs and outputs, we can start to consider the characteristics of this model that give rise to various testing challenges.

3.1.3 Characteristics of the Model

Although a relatively simple form of computational model, EBMs possess a number of characteristics that make them difficult subjects to test. These characteristics are particularly problematic for two testing activities: test case development and test oracle construction [13].

Regarding test case development, it can be difficult to identify appropriate input values and domain boundaries at which critical behaviour changes of interest occur [13]. This problem stems from the complexity of the subjects that are typically simulated by EBMs, such as disease outbreaks, for which the process of converting the underlying theory to software is non-trivial [126]. Practically speaking, this means it can be difficult to select input configurations that will evoke an anticipated response in output. This problem is partially mitigated in this case by the availability of historical data, as this enables suitable input parameterisations to be analytically determined.

The converse process of determining whether, given some inputs, a model implementation produces correct outputs - that is, to provide a test oracle - can also be difficult for EBMs [13]. In many situations, model developers turn to historical data as a form of test oracle. However, computational models are often used to explore uncertain phenomena for which the correct behaviour is typically unknown [13]. Without access to data that shows the expected behaviour of the subject matter, modellers may resort to professional judgement as a means for validating the outputs of modelling code instead [126]. For more complex subject matter, though, this approach is likely to be impractical or even impossible; such programs are sometimes referred to as being *non-testable* [30] (see Section 2.1.4).

Although historical data can partially alleviate the challenge of constructing test cases and test oracles in this case, this approach has two fundamental problems.

First, historical data can contain many forms of bias. Indeed, Ozmen et al. [121] state that data used to derive the initial parameters of the Spanish Influenza EBM are likely subject to biases that arises from the under-reporting of incidents and long-term fatality rates, as well as the methods used for data collection during the actual pandemic being impacted by demographic and geographic factors. This serves as evidence that, even when historical data is available, it's rarely perfect and so doesn't represent a perfect test oracle.

Second, historical data gives an account of what *actually* happened. However, one of the main strengths of EBMs and computational models in general is their ability to evaluate *what if?* scenarios; that is, to predict what *might have* happened. For example, a modeller might want to model how the cumulative infections in the Spanish Influenza pandemic would have changed, had the population had less daily contacts on average. We do not possess historical records for this alternative version of the pandemic and therefore do not know what the correct output should look like.

We summarise these challenges as follows:

Although EBMs are a relatively simple and efficient form of model, they remain difficult subjects to test. This difficulty is a result of the often complex subject matter simulated by EBMs, such as epidemics. The 'correct' behaviour of such complicated phenomena is sometimes unknown or poorly understood, and the precise input values that should give rise to particular behaviours are often unknown. Together, these problems make it difficult to develop test oracles and test cases.

These limitations motivate the need for testing techniques that do not depend on the availability of a full test oracle (i.e. given the inputs of a test case, knowledge of precisely what the correct output(s) should look like) and that can generate test cases that evoke an expected response. To this end, in Section 3.3, we demonstrate how metamorphic testing [16] can be applied to test this model using the metamorphic relations originally outlined by Pullum et al. [5]. However, before we apply metamorphic testing to this model, in the following section, we introduce a more advanced form of computational model that is also commonly used for epidemiological modelling and discuss how its more intricate characteristics give rise to a considerably more problematic testing challenge.

3.2 Motivating Example II: Epidemiological Agent-Based Model

Agent-Based Models (ABMs) are a form of computational model that operate at a finer level of detail than EBMs. One of the main factors that differentiates an ABM from an EBM is the explicit modelling of individual-level behaviour. In this way, the individuals of an ABM are *heterogeneous*, whereas those of an EBM are *homogeneous*. This heterogeneity means that agents have different characteristics that may influence both their individual behaviour and emergent, system-level behaviour. It is this ability to capture heterogeneous interactions that produces more realistic and informative simulations [120].

In this section, we begin by introducing our motivating example of an ABM, Covasim. We then provide a brief overview of its core mechanistic behaviour, with the aim of highlighting aspects of the modelling infrastructure and design that pose challenges for testing. In particular, we focus on the overlapping attributes of Covasim and the EBM from the previous section and draw a comparison between the complexity of both forms of model and the testing challenges they present.

3.2.1 The Model

As an example that is comparable to the EBM of Section 3.1, consider the popular open-source COVID-19 modelling tool, Covasim [127, 128]. Covasim is an epidemiological ABM that has been used to inform COVID-19 policy decisions in several countries [129, 130, 131, 132]. The primary use case of Covasim is to evaluate the effect of various interventions, such as vaccines and social distancing guidelines, on the spread of COVID-19 in order to help policymakers to make informed decisions [127].

We selected this model as it is also a model of infectious disease that is testable against the same set of metamorphic relations originally designed for the previous EBM (see Section 3.1). This provides us with a systematic means to compare the suitability of metamorphic testing for two different forms of computational model that present different testing challenges. Moreover, epidemiologists are increasingly turning to individual-level models such as ABMs for modelling disease outbreaks [133]. Covasim therefore serves as a timely and more representative example of the sort of modelling software that is currently being used in practice.

3.2.2 Model Details

We now provide a high-level overview of the mechanistic behaviour of Covasim, covering three key functions of the model: (i) disease progression, (ii) viral transmission, and (iii) contact networks. For a more detailed overview, we refer the reader to Kerr et al.'s paper on the methodology underpinning Covasim [127]. Our objective here is to highlight aspects of the model design and implementation that ultimately hinder the application of state-of-the-art testing techniques, such as metamorphic testing. Most notably, we highlight the several ways in which the model introduces complex, non-linear behaviour and non-determinism; two aspects which increase the overall difficulty and cost associated with testing.

Disease Progression

As with the Spanish Influenza EBM (Section 3.1), the disease progression model in Covasim simulates the transition of individuals through different disease states. However, Covasim splits the infectious state into five separate states of disease severity: presymptomatic, asymptomatic, mild, severe, and critical. The full set of states modelled in Covasim is summarised in Table 3.3.

Unlike the in Spanish Influenza EBM, the transition of individuals through disease states in Covasim is not parameterised by a fixed set of parameters and equations, but is governed by a stochastic process that accounts for individual characteristics, such as age and viral load. For example, the susceptibility of individuals to disease increases with age, meaning that older individuals are more likely to be infected after an infectious contact.

The key difference here is that in Covasim, disease progression is a dynamic process in which some characteristics of individuals, such as their propensity to infect others, change alongside their disease state, while others influence the disease progression itself. This makes disease progression a difficult mechanism to test as the expected behaviour will vary depending on characteristics of the population. This greatly increase the challenge of identifying the 'correct' behaviour for a given scenario.

Variable	Symbol	Description
Susceptible	S	The number of individuals susceptible to infection.
Exposed	E	The number of infected individuals that are not yet infectious.
Presymptomatic	I_P	The number of infectious individuals that do not yet display symptoms.
Asymptomatic	I_A	The number of infectious individuals that display no symptoms.
Mild	I_M	The number of infectious individuals that have mild symptoms.
Severe	I_S	The number of infectious individuals that have severe symptoms
Critical	I_C	The number of infectious individuals that have critical symptoms.
Recovered	R	The number of infected individuals that have recovered.
Dead	D	The number of infected individuals that have died.

Table 3.3: A description of the state variables that are modelled in the Covasim ABM.

Viral Transmission

The second essential feature of Covasim is viral transmission. This is the process through which the virus is transmitted upon contact between susceptible and infectious individuals. Unlike in the Spanish Influenza EBM (Section 3.1), the expected number of daily transmissions is not a shared constant value for all contacts. Rather, it varies from contact to contact as it depends on a number of factors, such as the viral load and susceptibility of the individuals involved.

The process of viral transmission is complicated further by the presence of interventions. For example, vaccines can be implemented that boost the antibody level of an individual which, in turn, protect them from serious illness. Interventions are specified as complex objects that are configured by several parameters, adding to the already vast and complex input space of Covasim.

Another challenge surrounding viral transmission is uncertainty in inputs. Specifically, many of the input parameters in Covasim are captured by probability distributions that incorporate a large degree of uncertainty. For example, individual viral load is drawn from a negative binomial distribution [127]. This means that two otherwise identical individuals may be assigned different values for different attributes that influence disease progression. This uncertainty introduces non-deterministic behaviour, meaning a given input configuration can produce different results from execution to execution.

Contact Network

In contrast to the Spanish Influenza EBM, Covasim includes an explicit model of the different environments in which a contact may occur. Covasim refers to these environments as ‘contact layers’ and, collectively, they form a ‘contact network’.

There are three pre-configured contact networks available in Covasim: ‘random’, ‘synthetic’, and ‘hybrid’. At the most simplistic end of the scale, the ‘random’ contact network places all agents into a single layer with a fixed number of daily contacts. At the most realistic end of the scale, the ‘synthetic’ contact network constructs high fidelity environments from real-world data, such as census data. As a compromise, the ‘hybrid’ contact network generates four contact layers (household, workplace, school, and community) using location-specific age and household contact distribution data. We use the hybrid setting throughout this thesis as it is a compromise between the efficiency of random populations and the fidelity of synthetic populations.

In Covasim, agents can be assigned to contact layers based on characteristics such as age. For example, while all agents are assigned to the community layer on the ‘hybrid’ setting, agents of working age (between 22 and 65, by default) are additionally assigned to the workplace. The average number of contacts within the community layer is 20 per agent per day, whereas the average number in the workplace is 16. This is yet another avenue through which Covasim can exhibit complex, non-linear behaviour.

We summarise the aforementioned challenges imposed by Covasim’s more complex mechanics as follows:

In contrast to the Spanish Influenza EBM, the Covasim ABM follows a significantly more complex methodology that depends on individual-level characteristics, such as age and viral load. While this methodology greatly increases the realism and potential applications of the model, it also introduces non-determinism and complex non-linear behaviour. These factors significantly increase computational overhead.

3.2.3 Comparable Parameters

Now that we have an understanding of the core mechanistic processes behind Covasim, we can describe the key parameters involved, which are described alongside their initial values in Table 3.4. Since Covasim has a vast and complex input space comprising 64 input parameters (27 of which are complex objects that are characterised by further parameters), we limit our discussion to only the parameters that feature in both models. It is therefore necessary to emphasise that Table 3.4 provides a simplified view of the input space for Covasim.

The initial values we select for our Covasim simulations are the default values as of version 3.0.7. The only exceptions are: the population size, initial infectious individuals, and the location, which are set to values used in the EBM to facilitate a more accurate comparison. In addition, the population type which is set to ‘hybrid’, enables the use of contact layers (i.e. separate home, workplace, school, and community environments). Note that we do not modify parameters related to the disease itself (e.g. transmissibility), as we are not attempting to implement Influenza in Covasim. Rather, we are interested in the more general invariant properties of infectious diseases.

Table 3.4 also highlights the more complex interface of Covasim. For example, we can see that several of the parameters are described by parametric statistical distributions, such as death time and recovery time. In the case of the log normal distribution, this means two parameters (mean and standard deviation)

Variable	Symbol	Description	Initial value
Incubation time	σ	Time before an incubating individual becomes infectious.	*lognormal* $(4.5, 1.5)$
Death probability	α	The probability that an infected individual will be killed by the disease, given their age.	$\begin{cases} 0.667, & \text{if } 0 \leq \text{age} < 10 \\ 0.250, & \text{if } 10 \leq \text{age} < 20 \\ & \vdots \\ 0.476, & \text{if } 80 \leq \text{age} < 90 \\ 0.929, & \text{if age} \geq 90 \end{cases}$
Death time	γ_D	The time for death to occur for individuals killed by the disease.	*lognormal* $(10.7, 4.8)$
Recovery time	γ_R	The time for recovery for individuals who survive the disease, given the severity of their symptoms.	$\begin{cases} \textit{lognormal}\,(8,2) & \text{if asymptomatic or mild} \\ \textit{lognormal}\,(18.1, 6.3) & \text{if severe or critical} \end{cases}$
Transmissibility	τ	The probability that a susceptible individual is infected upon contact with an infectious individual.	0.016
Mean contacts	μ	The average number of daily contacts per individual per contact layer.	$\begin{cases} 2 & \text{in household layer} \\ 20 & \text{in school layer} \\ 16 & \text{in workplace layer} \\ 20 & \text{in community layer} \end{cases}$
Population size	N	The total number of living individuals in the model at a particular time.	103267000
Initial infectious	I_0	The number of infectious individuals at the point of initialisation.	1000

Table 3.4: A description of the parameters used in Covasim that are comparable to the Spanish Influenza EBM.

need to specified. We can also see that mean contacts and death probability are conditional in nature, with the assigned value being moderated by individual-level attributes such as age.

3.2.4 Model Outputs

Running Covasim with the parameters listed in Table 3.4 for a simulated duration of 200 days produces the time series shown in Figure 3.3. By comparing these time series to those produced by the EBM shown in Figure 3.1, we can see that both models exhibit broadly similar trends. That is, the rough shape of each curve is the same. However, the exact timing and magnitude of the various peaks and troughs differ. This is as expected, given that one model is simulating Influenza and the other is simulating COVID-19 and

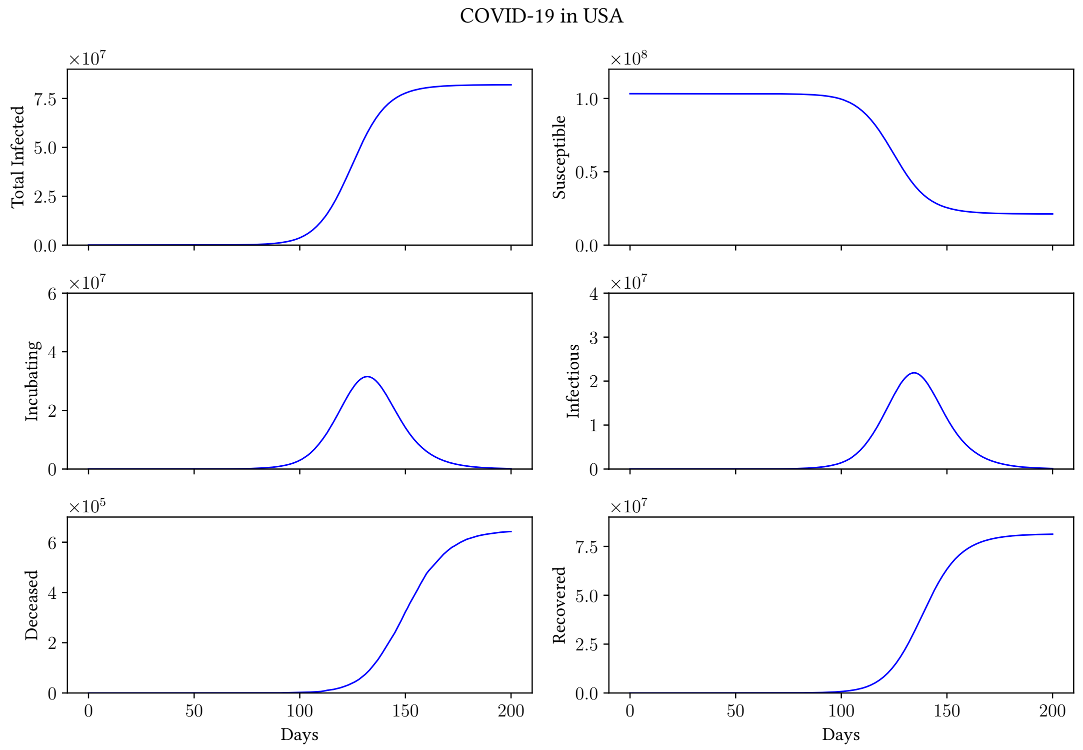


Figure 3.3: Time series showing the flow of agents through infectious disease states in Covasim as simulated using the parameters listed in Table 3.4.

that these viruses have different characteristics that influence the rate at which the resulting pandemic unfolds. For example, the highest risk population for the two diseases is known to differ: for COVID-19, elderly individuals over the age of 65 are at the highest risk of mortality, whereas for Spanish Influenza, the most vulnerable individuals were those aged between 25 and 40 years [134]. Such differences will inevitably lead to different behaviour.

Having now described the various mechanisms that underpin Covasim, as well as its inputs and outputs, we can discuss the different properties that pose difficulties to testing activities.

3.2.5 Characteristics of Agent-Based Models

In comparison to the EBM presented in Section 3.1, Covasim has a number of complex characteristics that pose a significant problem to state-of-the-art testing techniques, such as metamorphic testing.

First, Covasim has an expansive input space that includes complex inputs, such as interventions, that can be customised using many parameters. The sheer size and complexity of this input space makes adequate coverage of all input-output relationships unattainable in practice.

Moreover, many of the input parameters in Covasim are subject to large uncertainties [127]. This means that, to simulate any given scenario, a range of possible values must be considered, greatly increasing the cost and difficulty of testing. Further to this, as an ABM, Covasim exhibits behaviour at several different levels of abstraction, from agent-level to system-level, all of which should be considered during testing [34]. This once again increases the size and complexity of the problem space associated with testing ABMs such as Covasim.

In addition, while Covasim offers a higher level of model fidelity, this comes at a cost. Under realistic settings, simulations in Covasim can have long execution times and high computational costs, with non-trivial runs taking hours and accumulating large amounts of data. This limits the applicability of testing techniques that rely on repeated executions of the SUT.

To compound this issue further, Covasim is also non-deterministic: running the same simulation parameters multiple times (with a different seed) will yield different results. This means that each test case should be executed several times to observe a distribution of outcomes, significantly increasing the computational overhead of the already time consuming and costly testing process.

Aside from the nuances of the software itself, the complex subject matter of Covasim also presents a challenge for testing: specifically, the test oracle problem [15]. As discussed in our overview of the Spanish Influenza EBM (Section 3.1), given the exploratory setting that Covasim is typically deployed in, for most modelling scenarios, the precise expected output is unknown. This makes Covasim a traditionally "untestable" [30] system as the expected output is seldom known, making it difficult to determine whether the output of a given test is correct.

Agent-based models, like Covasim, have a number of challenging properties that make them exceptionally difficult subjects to test. These properties include (i) their long execution times and high computational costs; (ii) the vast and uncertain input space; (iii) non-deterministic behaviour; and (iv) exploratory mode of operation which gives rise to the test oracle problem.

3.3 Metamorphic Testing of Epidemiological Models

As a result of the test oracle problem, conventional testing methods are difficult to apply to computational models. One solution that is commonly used to alleviate the test oracle problem is *metamorphic testing* [13, 16], a form of testing in which properties are defined over the inputs and outputs of the SUT that express how the output should change in response to a particular change in some inputs [135]. Metamorphic testing is a powerful solution to the test oracle problem as it enables one to show correctness with respect to particular properties in the absence of a full test oracle. For a more in-depth introduction to metamorphic testing, refer to Section 2.1.5.

In this section, we demonstrate how metamorphic testing can be applied to both of the motivating examples and discuss the extent to which it addresses the challenging characteristics introduced earlier in this chapter. Our implementation can be found in our open source repository[2].

3.3.1 Metamorphic Testing: Equation-Based Model

We begin with our implementation of the 1918 Spanish Influenza EBM originally introduced by Sukumar et al. [3], for which Pullum and Ozmen later defined a series of epidemiological metamorphic relations

[2] https://github.com/AndrewC19/phd_thesis_code_examples/tree/main

[5]. These metamorphic relations quantify the expected relationship between the various parameters and states, and are summarised in Table 3.5 below. While fairly broad in nature, they provide a means for us to check that our implementation is not only accurate with respect to historical data, but that the implementation responds in a logically correct way to changes in the parameters.

For example, MR5 in Table 3.5 states that decreasing the amount of time death takes to occur should reduce the total infections and deaths. This follows from the fact that dying individuals are still infectious, so removing them from the model sooner reduces the likelihood that they spread the disease. Collectively, the 14 metamorphic relations specified in Table 3.5 form a suite of properties that, once tested, give us confidence in the correctness of the implementation.

It is important to note that none of these metamorphic relations depend on access to historical data, nor are they specific to Spanish Influenza. Rather, they are general properties of infectious diseases that could, in principal, be transferred to another epidemiological model of an entirely different disease, with slight modifications to account for differences in interface (i.e. inputs and outputs) or implementation.

ID	Parameter	Relation	Expected outcome
MR1	α	$\alpha' = n\alpha; n < 1$	D should decrease by at most a factor of n and R should increase by at least $\frac{1}{n}$.
MR2	α	$\alpha' = n\alpha; n > 1$	R should decrease by at least $\frac{1}{n}$ and D should increase by a factor of at most n.
MR3	I_0	$I_0' = nI_0; n < 1$	The start of the rise in D should be delayed.
MR4	I_0	$I_0' = nI_0; n > 1$	D should plateau sooner.
MR5	γ_D	$\gamma_D' = n\gamma_D; n < 1$	T_I and D should decrease.
MR6	γ_D	$\gamma_D' = n\gamma_D; n > 1$	T_I and D should increase.
MR7	τ	$\tau' = n\tau; n < 1$	I, D and R should decrease.
MR8	τ	$\tau' = n\tau; n > 1$	I, D and R should increase.
MR9	σ	$\sigma' = n\sigma; n < 1$	The start and end of the rise in D should occur sooner.
MR10	σ	$\sigma' = n\sigma; n > 1$	The start and end of the rise in D should occur later.
MR11	γ_R	$\gamma_R' = n\gamma_R; n < 1$	I and D should decrease.
MR12	γ_R	$\gamma_R' = n\gamma_R; n > 1$	I and D should increase.
MR13	μ	$\mu' = n\mu; n < 1$	I and D should decrease.
MR14	μ	$\mu' = n\mu; n > 1$	I and D should increase.

Table 3.5: Metamorphic relations proposed by Pullum and Ozmen for a Spanish Influenza equation-based model [5].

Metamorphic Testing Setup

To demonstrate the application of metamorphic testing to EBMs, we implemented and executed metamorphic test cases for each of the metamorphic relations in Table 3.5. In keeping with the original paper, the source test case was configured using the set of parameter values derived from historical data, as shown in Table 3.2. While the original paper [5] appears to create follow-up test cases by hand-selecting a sensible factor to increase or decrease the input of interest by, we take a systematic approach in which we randomly sample 30 distinct factors according to the metamorphic relation. To achieve this, for increasing an input value, we sampled a factor from the uniform distribution $U(1,30)$. Similarly, for decreasing an input value, we sampled a factor from the uniform distribution $U(0,0.999)$. We selected the upper and lower bounds of these distributions to prevent our metamorphic test cases from making extremely large and entirely unrealistic changes to the input value.

As an example, for MR5, we formed a follow-up test case in which the mortality duration (γ_D) was decreased (multiplied by a factor of $n < 1$) while all other inputs remained unchanged. We then implemented the appropriate assertions to check whether the expected outcomes held. In this case, the assertion simply checked whether the cumulative infections and deaths on the final day of simulation decreased from the source to the follow-up execution. This process was repeated 30 times (each time sampling a fresh multiplicative factor to perform distinct metamorphic tests of the same relation) for each metamorphic relation and we report the associated pass rate as the proportion of the 30 tests that pass, as well as the mean wall clock time for execution.[3]

Metamorphic Test Results

The results of testing our implementation of Sukumar and Nutaro's Spanish Influenza EBM [3] against Pullum and Ozmen's metamorphic relations [5] are shown in Table 3.6.

The wall clock execution times shown in Table 3.6 demonstrate the computational efficiency of the Spanish Influenza EBM, with the 30 metamorphic tests for each metamorphic relation taking between 0.24 and 0.32 seconds to execute on average. However, while the majority of the metamorphic relations pass for all 30 tests, there are two exceptions: MR2 and MR3.

MR2 only passed 73% of the time. Looking into the tests that failed this metamorphic relation, we found that all of the failing tests had a mortality probability (α) greater than 0.2, and that all passing tests were below this threshold. Above this threshold, the mortality probability is so high that the outbreak is significantly slowed down. This occurs because many infected individuals die before they can transmit the disease. An example of this phenomenon is shown in Figure 3.4.

MR3 passed 93% of the time. On further inspection of the failing tests for this relation, we found that all failures occurred when the number of initially infectious individuals (I_0) had been reduced by a comparatively small amount (no more than 6%). This comparatively small reduction did not have a notable impact on the timescale of the outbreak, leading to the failure of the metamorphic relation, as shown in Figure 3.5, which is visually identical to the output of the source test case shown in Figure 3.1.

In both cases, the failure can be attributed to a poor specification of the input parameter boundaries - that is, the threshold at which some critical behaviour change occurs. This problem could be addressed by specifying a maximum increase in α for MR2 and a minimum decrease in I_0 for MR3. This aligns with Kanewala and Bieman's survey [13], which highlights the a priori identification of critical domain input boundaries as a key testing challenge in the scientific software domain.

[3]For a given metamorphic relation, we measure the wall clock time as the amount of time required to set-up and execute the model with both the source and follow-up test case. This is can be viewed as a worst case time necessary to execute each metamorphic relation independently as, due to the deterministic nature of the EBM, it is only necessary to execute the source test case once. These tests were executed on a virtual Linux machine running Ubuntu 20.04 with 30.5 GiB RAM, 4 Xeon E5-2686 v4 vCPUs, and a 6 GiB NVIDIA Tesla M60 GPU. We use the same machine to record all timings reported in this chapter.

		Execution Time (s)			
ID	Pass Rate	Minimum	Maximum	Mean	Total
MR1	1.00	0.0086	0.0105	0.0090	0.2687
MR2	**0.73**	0.0067	0.0109	0.0083	0.2500
MR3	**0.93**	0.0087	0.0092	0.0089	0.2663
MR4	1.00	0.0086	0.0095	0.0089	0.2673
MR5	1.00	0.0085	0.0095	0.0089	0.2672
MR6	1.00	0.0087	0.0100	0.0091	0.2721
MR7	1.00	0.0070	0.0090	0.0080	0.2414
MR8	1.00	0.0098	0.0113	0.0106	0.3182
MR9	1.00	0.0086	0.0143	0.0105	0.3153
MR10	1.00	0.0068	0.0084	0.0072	0.2157
MR11	1.00	0.0074	0.0119	0.0096	0.2888
MR12	1.00	0.0095	0.0103	0.0099	0.2958
MR13	1.00	0.0071	0.0122	0.0089	0.2679
MR14	1.00	0.0098	0.0118	0.0106	0.3186

Table 3.6: Results of metamorphic tests for our implementation of the 1918 Spanish Influenza EBM using metamorphic relations proposed by Pullum and Ozmen [5], showing the pass rate and execution time statistics for each metamorphic relation. Here, the total time is the time taken to execute 30 metamorphic tests, in which the source and follow-up test cases are executed one time each.

3.3.2 Metamorphic Testing: Agent-Based Model

We now turn our attention to our second motivating example: the Covasim ABM introduced in Section 3.2. In this section, we apply the same metamorphic relations outlined by Pullum and Ozmen [5] to Covasim to demonstrate how metamorphic testing can be applied to the significantly more complex nature of Covasim and highlight the additional challenges this presents. While these metamorphic relations were originally defined for an EBM of the 1918 Spanish Influenza pandemic, they capture general properties of infectious disease rather than properties specific to Spanish Influenza. For this reason, we expect the majority of the metamorphic relations to hold in Covasim, despite the change in modelling paradigm and disease being simulated. As a secondary goal, we test the extent to which this theory holds.

Metamorphic Testing Setup

For our source test case, we use the default Covasim parameter settings other than population size and initial infectious individuals, which are set to 103267000 and 1000, for consistency with the initial population of the Spanish Influenza model. We also use the 'hybrid' population type for a compromise between model fidelity and efficiency. As with the EBM, we form our follow-up test cases by multiplying the input of interest by a factor, which is sampled randomly from the uniform distributions $U(0, 0.999)$ for decreases and $U(1, 30)$ for increases.

On attempting to simulate these settings, we found encountered two problems. First, Covasim runs out of memory and crashes due to the large population size. To address this, we use a feature of Covasim known as 'population scaling', where each individual agent in the simulation is made to represent a specified number of identical individuals, enabling us to reduce the actual simulated population size

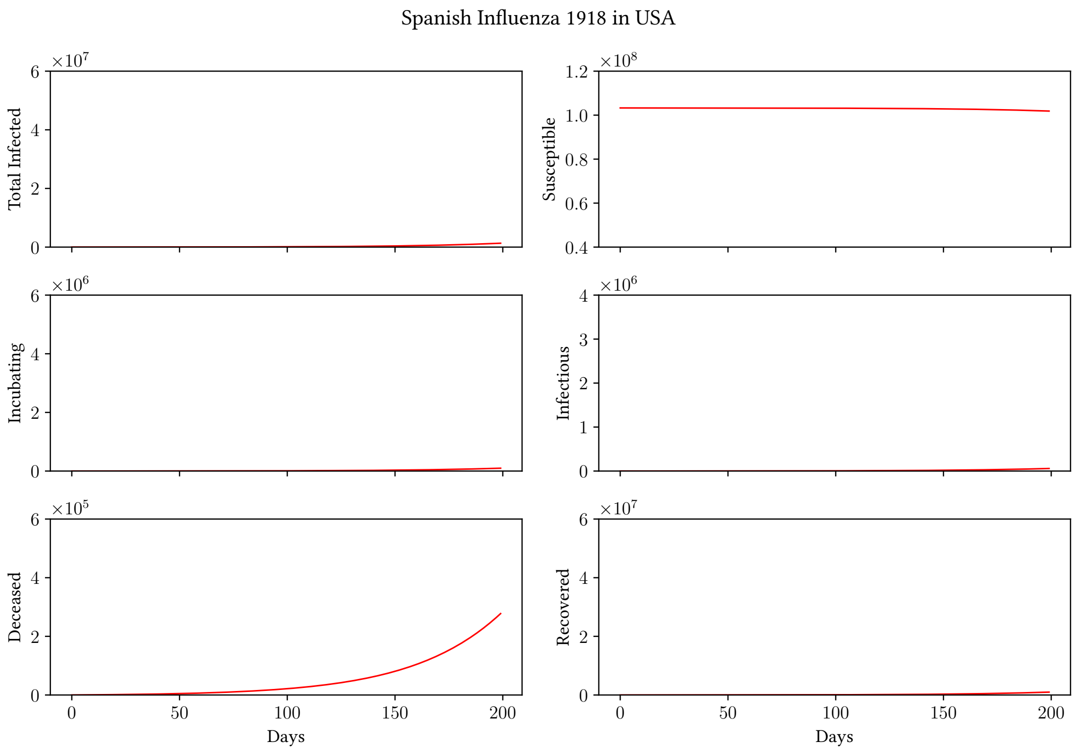


Figure 3.4: Spanish Influenza EBM with $\alpha = 0.22$, delaying the pandemic to such an extent that MR2 fails.

while obtaining results for the intended population size. For reasons of feasibility that we elaborate on in the following section, we use a scaling factor of 10000. This means every agent in the simulation represents 10000 identical individuals.

We work on the assumption that population scaling *does not* scale-up the specified initially infectious population, since our trial simulations show that the number of exposed (initially infectious) individuals on day zero matches the supplied value for this input parameter, even when a scaling parameter is applied (i.e. the scaling parameter does not appear to change the number of initial infections on day zero). However, the intended interaction between population scaling and the initially infectious population is not explained in the documentation for Covasim, leading to the second problem we encountered: population scaling appears to prevent us from increasing the initially infectious population (for $MR4$) by more than a factor of 10. Doing so leads to an error caused by attempting to initialise the model with an initial infectious population that is larger than the total population size.

Therefore, by applying $MR4$ to Covasim, we have used metamorphic testing to uncover a potential bug related to population scaling or result reporting in Covasim. On the one hand, if our assumption that population scaling should not apply to the initially infectious population is incorrect, then the reported number of exposed individuals on day zero should be scaled-up (this is not the case). We would then

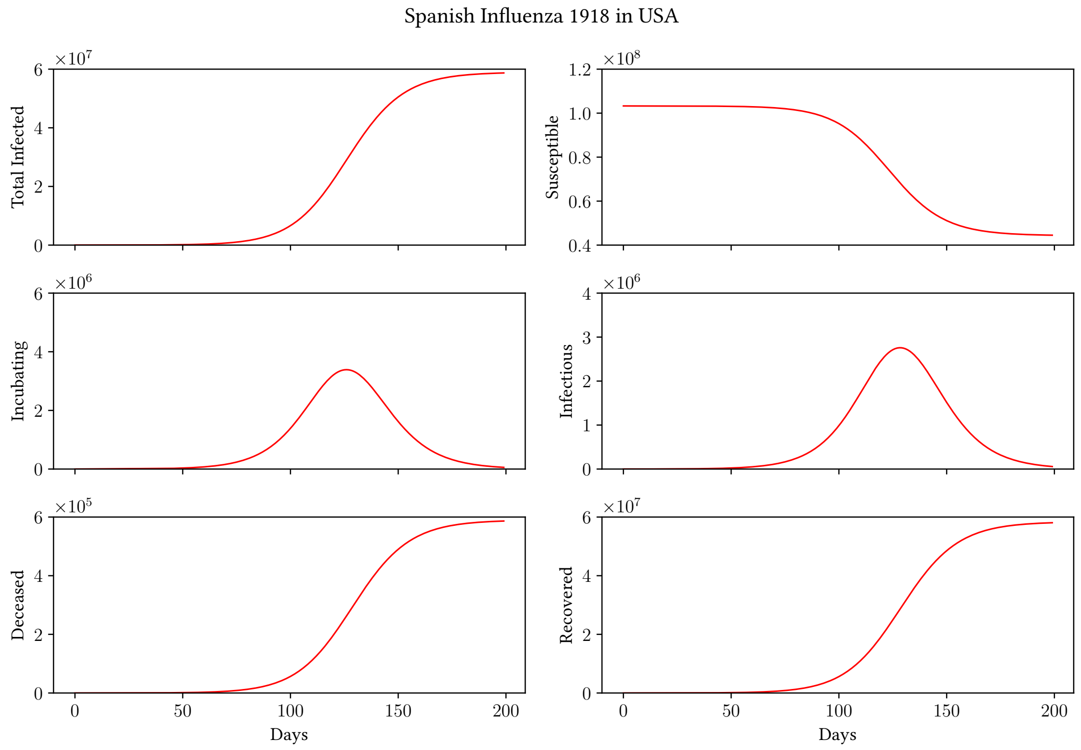


Figure 3.5: Spanish Influenza EBM with I_0 reduced from 1000 to 950. This small reduction does not produce a notable change in the duration of the outbreak, causing MR3 to fail.

reduce the initial infectious population size by the same factor to rectify this problem. On the other hand, if our assumption is correct (i.e. the scaling factor should not affect the initial infectious population), it should be possible to increase the initially infectious population by a factor of at least 30, as this would result in $1000 \times 30 = 30000$ initially infectious agents, which is significantly less than the full (scaled-up) population of 103267000. It is unclear what the root cause of this problem is.

We raised an issue on GitHub to flag this inconsistent behaviour as a potential bug to the developers[4]. We discuss this inconsistency here as it affects our ability to perform $MR4$ in a way that is consistent with its application to the EBM (Section 3.3.1). It is important to emphasise, however, that this problem is not a limitation of metamorphic testing, but is caused by attempting to run Covasim with the aforementioned settings. Fortunately, this issue does not prevent us from applying $MR4$ entirely, it merely limits the size of the increase in infectious population we can apply. It is important to note, however, as it limits our ability to draw a direct comparison.

Nonetheless, having now discussed the parameter settings used to configure our metamorphic tests, in the upcoming section, we outline how we adapted the metamorphic testing process to account for the non-deterministic nature of Covasim. That is, the changes made to perform the non-deterministic variant

[4]https://github.com/InstituteforDiseaseModeling/covasim/issues/397

of metamorphic testing: SMT (see Section 2.1.5) [53].

Handling Covasim's Non-Deterministic Behaviour

Due to the non-deterministic nature of Covasim, we cannot apply metamorphic testing in exactly the same way as we did to the EBM. Instead, we need to observe a distribution of outcomes for both the source and follow-up test case for every metamorphic relation. To this end, in this section, we use SMT in place of conventional metamorphic testing. The essential difference here is that, for each metamorphic relation, both the source and follow-up test cases are repeated multiple times with different seeds to obtain a pair of distributions of outputs: one for the source test case executions and another for the follow-up test case executions. We then compute and contrast the means of these distributions to obtain the average change in output. We perform 30 repeats to obtain our distributions as a balance between execution time and variability in the output.

As before, for each metamorphic relation we generate and perform 30 distinct metamorphic test cases - that is, 30 metamorphic tests in which the follow-up test cases are all different. Note that, for each of these 30 distinct metamorphic test cases, the source and follow-up test cases are executed 30 times each, as described above. We refer to this latter process as 'repeating' the metamorphic tests. This means that testing an individual metamorphic relation with 30 repeats involves $30 \times 30 \times 2 = 1800$ executions. Given this significant number of executions, it is important to consider the execution time of each test.

Initially, we set the population scaling value to 10, which allowed an individual simulation (of the source test case) to be executed in roughly 18 minutes. Assuming we repeat each metamorphic test 30 times to account for non-determinism, this means:

- An individual test for one metamorphic relation would require roughly **18 hours** to execute.
- 30 distinct metamorphic tests for one metamorphic relation would require around **540 hours** to execute.
- 30 distinct metamorphic tests for all 14 metamorphic relations would require approximately **7560 hours (315 days)** to execute.

While this is a rough approximation and means such as parallelisation can be employed to drastically reduce the overhead, it illustrates an important point: *computational modelling software can be expensive and time-consuming to run making the application of state-of-the-art testing techniques, such as SMT, potentially infeasible*. Furthermore, the approximate computation times quoted above are based on simulations with a scaling factor of 10 and a hybrid population. Running the same simulation with no scaling factor caused a memory error. However, had it been possible to run the simulation with these parameters, the computation time would be even higher.

It is worth clarifying here that this computational overhead should not be attributed to metamorphic testing alone, rather it is predominantly caused by the combination of non-determinism and long execution times associated with complex computational models. Specifically, for n metamorphic relations, the non-determinism requires $30n$ test executions (assuming 30 repeats); metamorphic testing, on the other hand, requires either $(n + 1)$ or $2n$ test executions, depending on whether the source test case is re-used or not.

Nonetheless, with these costs in mind, for the purpose of this chapter, we perform metamorphic testing as described above but using a population scaling setting of 10000, meaning each individual represents

10000 identical individuals. This setting drastically reduces the run-time of each simulation of the source test case to a matter of seconds, enabling us to test the 14 metamorphic relations in a feasible amount of time. It is worth noting, however, that population scaling presents a trade-off between efficiency and model fidelity. Specifically, population scaling discretises the results into increments of the scaling factor (10000 in our case), which reduces the precision of the outputs [127].

Given the coarse nature of the metamorphic relations we are testing, this decrease in model fidelity is unlikely to impact test outcomes. More specifically, the metamorphic relations are coarse in the sense that they are not concerned with the precise change in infections, deaths, etc., but whether these outputs increase, decrease, or are delayed. Thus, for the purpose of demonstrating the application of SMT to Covasim, this decrease in model fidelity is feasible. However, in situations where the precision of the outputs is essential (e.g. when predicting the effect of interventions such as vaccines), model fidelity would take precedence and therefore the computation time associated with simulation and testing would be significantly higher. In Chapter 6, we address this dependence of metamorphic testing on many costly executions using graphical causal inference methods to predict metamorphic test outcomes from existing, passively observed testing data.

Therefore, while our results are obtained in a comparatively low amount of time, it is worth bearing in mind that real-world situations would inevitably present a more significant challenge in terms of scalability and cost. In the following section, we present the results of testing Covasim against the 14 metamorphic relations from Table 3.5 using the previously described SMT process.

Metamorphic Test Results

The results of testing Covasim against the metamorphic relations defined by Pullum and Ozmen [5] are shown in Table 3.7. As a reminder, the total run time here reflects the time required to execute 30 distinct metamorphic tests (i.e. each one using a different follow-up test case) and repeat this process 30 times to account for non-determinism. The pass rate denotes the proportion of the 30 distinct test cases that passed (i.e. the difference in the mean outcome of the source and follow-up executions followed the metamorphic relation).

The wall clock execution times shown in Table 3.7 demonstrate the greater computational cost associated with Covasim, even when using a population scaling factor of 10000. In total, it took roughly 14 hours of computation time to test all 14 metamorphic relations using 30 distinct metamorphic tests, with each being repeated 30 times. In the most extreme case, we can see that $MR14$ took a total of 6179.8032 seconds to execute. This high execution time is the result of increasing the number of contacts, as prescribed by $MR14$, and demonstrates another avenue through which the cost of metamorphic testing can be affected: the choice and magnitude of change in input.

In addition, Table 3.7 reveals that several metamorphic relations did not pass for all 30 metamorphic tests.

Most notably, $MR1$ and $MR2$ failed all 30 metamorphic tests. This outcome can be explained by a simple difference in implementation between the EBM described in Section 3.1 and Covasim. In the differential equations of the EBM that govern the change in deaths and recoveries (Equations (3.1.4) and (3.1.5)), we can see that death and recovery are both controlled by a single parameter: α for the probability of death and $(1 - \alpha)$ for the probability of recovery. In contrast, Covasim parameterises death and recovery separately and therefore changes to the probability of death have no direct effect on

		Execution Time (s)			
ID	Pass Rate	Minimum	Maximum	Mean	Total
MR1	**0.00**	97.6178	102.9877	100.7494	3022.4822
MR2	**0.00**	101.7086	105.3109	103.3813	3101.4402
MR3	**0.87**	107.9961	111.8038	109.7768	3293.3045
MR4	1.00	107.5028	111.5559	109.5428	3286.2835
MR5	**0.33**	111.6074	115.7829	113.6222	3408.6655
MR6	**0.03**	115.0924	117.7206	116.6207	3498.6218
MR7	**0.97**	98.4862	123.6635	110.7651	3322.9522
MR8	1.00	104.0057	109.5601	106.1269	3183.8075
MR9	1.00	115.5047	125.4498	119.3094	3579.2809
MR10	1.00	105.5033	129.4997	112.9865	3389.5949
MR11	**0.97**	109.5847	131.5789	123.6047	3708.1413
MR12	1.00	127.9719	139.0556	134.3034	4029.1024
MR13	**0.93**	104.6890	140.8631	124.8228	3744.6848
MR14	1.00	139.5898	297.1204	205.9934	6179.8032

Table 3.7: Results of applying the metamorphic tests for the metamorphic relations proposed by Pullum and Ozmen [5] to Covasim, showing the pass rate and execution time statistics for each metamorphic relation. Here, the total time is the time taken to execute 30 distinct metamorphic tests (i.e. each one using a different follow-up test), with each distinct metamorphic test being repeated 30 times to account for non-determinism.

recoveries. Consequently, the executions of $MR1$ and $MR2$ are extremely unlikely to observe a change in recoveries that surpasses the threshold of a factor of at least $\frac{1}{n}$, causing them to almost certainly fail[5].

We confirmed this theory by altering $MR1$ and $MR2$ to remove the second condition that recoveries should increase/decrease by a factor of at least $\frac{1}{n}$, respectively. The results of testing the revised metamorphic relations are shown in Table 3.8, with both now achieving a 100% pass rate.

		Execution Time (s)			
ID	Pass Rate	Minimum	Maximum	Mean	Total
Covasim MR1	1.00	99.0719	106.4617	100.7463	3022.3902
Covasim MR2	1.00	104.3321	107.8105	106.0462	3181.3864

Table 3.8: Results for revised versions of $MR1$ and $MR2$ that capture the anticipated effect of changes to the probability of death on the number of deaths only to capture the behaviour of Covasim.

Table 3.7 also shows that $MR5$ and $MR6$ failed most of the time. This outcome is a result of the more complex transmission model used by Covasim that we outlined in Section 3.2.2. Specifically, in Covasim, infectious agents are separated into further compartments based on the severity of their symptoms (mild, severe, or critical). Within each of these compartments, various parameters and distributions are

[5]Differences in simulation that arise from non-determinism could, in theory, bring about a sufficiently large change in recoveries that would pass this metamorphic relation. However, this would be extremely unlikely.

modified according to disease severity (e.g. decreasing the probability of recovery as symptoms become more severe). However, only agents who are in the critical compartment can die. This means that the changes to death duration mandated by $MR5$ and $MR6$ only effect a minority of the population and thus any resulting change in infections or deaths is likely to be insignificant, making it difficult to distinguish from general noise caused by non-determinism.

Table 3.7 also reveals that $MR3$, $MR7$, $MR11$, and $MR13$ occasionally failed (less than 15% of the test executions). Upon examining the failing test runs, we can see that the test failures for these metamorphic relations occurred when the follow-up test case made an insufficient change to the input variable. The resulting changes in output are insignificant and can therefore be overpowered by changes in the same output caused by general non-deterministic factors that vary between the source and follow-up execution. For $MR7$, $MR11$, and $MR13$, all failures occurred when the input's value was changed by less than 10%. For $MR3$, one failure occurred with a change in input as large as 25%, despite several test cases passing that made much smaller changes to the input. This reflects an additional challenge introduced by non-determinism: metamorphic tests may pass and fail seemingly spuriously due to changes in output caused by uncontrolled non-determinism.

3.3.3 Limitations of Metamorphic Testing for Computational Models

Now that we have applied metamorphic testing to both of our motivating examples, we draw upon the results and findings of the previous sections to discuss the suitability of metamorphic testing as a solution to the test oracle problem for computational modelling software. In particular, we discuss how metamorphic testing adapts to the challenging properties associated with computational models that we previously outlined in Sections 3.1 and 3.2, highlighting several key limitations that hinder its application in practice.

Test Case Construction

In Section 3.1, we discussed how the complex subject-matter often simulated by EBMs can make it difficult to construct test cases - a point that has also been made in Kanewala and Bieman's survey on testing scientific software [13]. To this end, one of the benefits afforded by metamorphic testing is the convenient test case generation mechanism built into metamorphic relations: they outline a procedure that can be followed to transform arbitrary source test cases into follow-up test cases. In this way, metamorphic testing is already known to assist with test case construction.

However, as stated in Kanewala and Bieman's survey, one of the main barriers to developing test cases for scientific software, such as computational models, is the identification of input boundaries at which critical behaviour changes occur. In other words, it is difficult to identify values for inputs that cause the behaviour of the SUT to change in a significant, noticeable way,. In the context of metamorphic testing, this can make it difficult to appropriately scope or refine metamorphic relations.

When testing the metamorphic relations outlined by Pullum et al. [5] on the Spanish Influenza EBM in Section 3.3.1, we observed evidence of this problem first-hand: two of the metamorphic relations, $MR2$ and $MR3$, failed despite performing valid metamorphic tests (i.e. tests that changed the inputs of interest by a factor that was permitted by the metamorphic relation). Further inspection of these test failures revealed that they occurred when the changes made to the parameter were either too big or too small, causing the EBM to fall outside the intended 'operating window' of the metamorphic test.

For example, when the probability of death (α) was increased above 0.2 in the failing tests for $MR2$, the behaviour of the model changed significantly: death occurred with such high probability that the majority of the infectious population would die, causing the pandemic to slow down or peter out completely, rather than causing an increase in deaths (and decrease in recoveries) as asserted by the metamorphic relation. The problem here can be attributed to a misspecification of appropriate domain boundaries, i.e. rather than permitting *any* increase in α, $MR2$ should permit only increases that do not cause α to exceed the threshold of 0.2[6].

Test Oracle Problem

In Section 3.1, we also described how EBMs can suffer from the test oracle problem [15]. That is, the process of distinguishing whether the outputs corresponding to the inputs of a test case are correct or incorrect can be difficult. Here we explained that historical data is sometimes used as a form of test oracle (i.e. to determine whether a simulation accurately reproduces historical accounts), but noted that historical data has several key limitations that limit its suitability. Most notably, historical data can be helpful if our aim is to simulate a pandemic exactly as it occurred (e.g. to recreate the 1918 Spanish Influenza pandemic), but is less helpful if we want to employ our model more flexibly to investigate how a phenomenon behaves in various different hypothetical scenarios (e.g. to predict the effect of interventions on the spread of COVID-19).

For this latter ‘exploratory’ mode of operation, which is arguably the primary use case for most computational modelling software (certainly for epidemiological modelling software), it is necessary to ascertain the correctness of the modelling software beyond scenarios that have actually been observed. However, given the aim is to explore unknown phenomena, it can be difficult to characterise what correctness looks like.

To this end, the primary advantage of metamorphic testing is that it enables software to be tested without having to know the correct outputs corresponding to some inputs, effectively circumventing the test oracle problem. For this reason, metamorphic testing is widely recognised as one of the the state-of-the-art techniques for testing so-called ‘non-testable’ software [30], such as computational models that have exploratory applications [13].

However, metamorphic testing does not entirely alleviate the test oracle problem. Rather, it reduces the problem to the identification of metamorphic relations: properties defined over pairs of executions of the PUT in terms of its inputs and outputs, such as those in Table 3.5. Such properties are generally domain specific in nature and therefore typically require specialist knowledge to curate. In Sections 3.1 and 3.2, we were fortunate as the metamorphic relations for the epidemiological models were already available. Nonetheless, this highlights one of the key challenges surrounding metamorphic testing: the task of identifying metamorphic relations.

While this is perhaps less of an issue in the domain of computational modelling, where the developers and users of the software are more likely to have a relevant scientific background and the necessary domain expertise [13], it still represents a significant barrier to entry.

[6]The precise value of α at which the expected behaviour changes may be more precisely defined than 0.2. This is an approximate value based on the metamorphic tests failures we observed.

Test Execution Cost

In contrast to the comparatively simple EBM introduced in Section 3.1, the vastly more complex ABM introduced in Section 3.2 carries a significant computational overhead that drastically increases the cost of metamorphic testing. To compound this cost further, the ABM is non-deterministic and therefore requires the use of SMT rather than conventional metamorphic testing. As shown in Section 3.3.2, this means that every metamorphic test must repeatedly execute the source and follow-up test cases to obtain a distribution of outputs. A measure of average, such as the mean, is then computed and compared against the metamorphic relation.

The sheer magnitude of this increase in cost becomes apparent when we compare the wall-clock execution times for metamorphic testing of the EBM shown in Table 3.6 with that of the ABM shown in Table 3.7. For example, while MR5 took less than half a second to test on the EBM, it took nearly an hour to test on the ABM (12757× longer). Furthermore, the execution times for the ABM were obtained using a large population scaling parameter (10000). This drastically reduced the execution time at the cost of model fidelity and thus the timings shown in Table 3.7 are not representative of the amount of time that would be required to test the model with higher fidelity settings.

In addition, we must consider the vastly more complex and expansive input space of Covasim and bear in mind that the 14 metamorphic relations tested in this section only scratch the surface. To be more exact, Covasim has 64 unique input parameters, some of which are complex objects like vaccines that are characterised by further input parameters. There are countless ways in which a complex input parameter, such as a vaccine, can be modified by a metamorphic test (e.g. changing the type of vaccine or number of doses given). Collectively, these issues mean coverage of even a small fraction of this input space would require an intractable amount of execution time.

3.4 Problem Statement

By applying metamorphic testing to the previous motivating examples, we have demonstrated some of the main challenges associated with testing complex, computational modelling software. Even for relatively simplistic forms of computational models, such as EBMs, we showed how metamorphic testing can be hindered by a number of limitations. We then applied SMT — the extension of metamorphic testing to non-deterministic subjects — to the more complex, non-deterministic ABM, Covasim. This demonstrated how the aforementioned limitations are exacerbated by the characteristics of higher fidelity modelling approaches, such as agent-based modelling, due to their non-deterministic nature paired with their significantly increased computational overhead.

With these challenges in mind, we now outline the problem statement for our thesis:

Problem statement
The applicability of metamorphic testing to computational models is currently hindered by two key factors: (1) the difficult and time-consuming process of constructing metamorphic relations, and (2) the execution cost resulting from the significant computational overhead associated with computational models.

Hence, there is a need for testing techniques that make metamorphic testing more sympathetic to the challenging properties of computational modelling software, and that are capable of systematically constructing metamorphic relations in a domain agnostic manner. We aim to address these problems in this thesis.

3.5 Concluding Remarks

In this chapter, we have introduced two forms of epidemiological computational model as motivating examples. We started by introducing a relatively simple EBM of the 1918 Spanish Influenza in Section 3.1, before moving onto a significantly more advanced example in the form of the Covasim ABM in Section 3.2. After giving an overview of the mechanistic procedures that underpin these models, we outlined a number of their challenging properties that give rise to notable testing challenges, including the test oracle problem and test case development.

In Section 3.3, we then demonstrated how metamorphic testing could be applied as a state-of-the-art solution to the test oracle problem using the metamorphic relations introduced by Pullum and Ozmen [5]. In addition, we explained and later demonstrated how SMT can be performed in place of conventional metamorphic testing in order to account for the non-deterministic nature of Covasim. We also highlighted the additional computational overhead that SMT incurs.

After conducting metamorphic testing, in Section 3.3.3, we drew on the results shown in Tables 3.6 and 3.7 to outline a number of limitations that surround the application of metamorphic testing to computational models. We focused our discussion on three key problem areas: (i) the difficulty of developing test cases, (ii) the test oracle problem, and (iii) the cost associated with executing metamorphic test cases (SMT test cases in particular).

With these limitations in mind, we then outlined the problem statement for this thesis in Section 3.4. As a reminder, this problem statement outlines the need to:

1. Assist computational modellers to identify metamorphic relations in a domain agnostic and systematic manner to help further alleviate the test oracle problem.
2. Reduce the cost associated with executing metamorphic test cases, particularly when dealing with non-deterministic systems.

In summary, this chapter has:

- Introduced two forms of epidemiological model as motivating examples: an EBM of the 1918 Spanish Influenza and an ABM of COVID-19.
- Highlighted challenging properties of these models that make various testing activities especially difficult/impractical, including test case development, test oracle development (the oracle problem), and test case execution.
- Performed metamorphic testing on these models, a common solution to the aforementioned problems, revealing a number of limitations that arise as a result of their challenging properties.
- Outlined the problem statement for this thesis with these limitations in mind.

Motivated by our problem statement, in the following chapter, we conclude Part I of this thesis by outlining a potential solution in the form of causal inference: a family of experimental and statistical techniques that specialise in establishing salient causal relationships. In particular, we argue that metamorphic testing and causal inference share a similar and fundamentally *causal* objective and, with this in mind, begin to outline a theoretical testing framework that facilitates the application of causal inference techniques to existing software testing problems, such as metamorphic testing. We eventually demonstrate how metamorphic testing can be expressed as a causal inference problem and outline the potential advantages of this formulation.

4 | Causal Testing Framework

As outlined in the previous chapter, computational modelling software often has challenging properties that make them an especially difficult class of software to test. Perhaps most notably, the complex and often exploratory subject matter of computational modelling software makes it hard or even impossible to establish what correct behaviour looks like, giving rise to the test oracle problem [15]. Metamorphic testing is generally regarded as a state-of-the-art solution to this problem. However, due to the usually long execution times and non-deterministic behaviour exhibited by computational modelling software, metamorphic testing can be prohibitively expensive in practice.

Motivated by these challenges, in this chapter, we turn our attention to the suitability of causal inference as a potential solution. To motivate this direction, in Section 4.1, we highlight the fundamentally causal objective of metamorphic testing and argue that it can be viewed and solved as a problem of causal inference, where the goal is to establish a causal effect and ultimately answer some causal question. In particular, we outline the potential of observational and graphical causal inference methods to address the two key challenges outlined in our problem statement (see Section 3.4) that currently hinder the application of metamorphic testing to computational modelling software.

After outlining our motivation for exploring the overlap between causal inference and metamorphic testing, in Section 4.2.2, we then extend Staats et al.'s **Software Testing Framework** (STF) to form the **Causal Testing Framework** (CTF): a conceptual software testing framework that frames causality-focused testing activities as problems of causal inference. Our objective here is to re-define the core components of the STF in a way that facilitates the direct application of causal inference methods to testing problems. In Section 4.2.3, we then contrast the CTF with the STF to highlight the nuances of a causal approach to testing. In Section 4.3, we conclude this chapter by expressing metamorphic testing as a causal inference problem using the components of the CTF and discussing the theoretical advantages offered by this formulation.

The CTF proposed in this chapter will form the basis of the contributions presented in Chapter 5 and Chapter 6, which leverage causal inference to address the challenges of identifying metamorphic relations and the cost of executing metamorphic test cases, respectively (as motivated by Chapter 3). While we focus on metamorphic testing throughout this thesis, we design the framework to encompass any testing activity that has a fundamentally causal objective.

This chapter is based on and extends the conceptual software testing framework introduced in publication **P3**: *Testing Causality in Scientific Modelling Software* (TOSEM, 2023).

4.1 Causal Inference and Metamorphic Testing

As introduced in Section 2.2, causal inference is a family of methodologies and techniques for quantifying causal relationships and answering causal questions. From a causal standpoint, metamorphic testing has a similarly causal objective: to determine whether particular changes to input configurations cause the model to produce an expected change in output. In this way, a metamorphic test can be thought of as a quasi-experiment for answering a causal question that concerns the fulfilment of a specific metamorphic relation.

For example, the metamorphic relation $sin(x) = sin(\pi - x)$ essentially asks the question *"Does changing the input of sin from x to $\pi - x$ cause its output to change?"* and states the expected outcome: *it should not*. It is then the aim of the metamorphic test to perform an experiment that measures the causal effect of interest in order to confirm or refute this hypothesis. That is, to perform an experiment that isolates and measures a causal effect: *causal inference*.

One of the main advantages of causal inference is its ability to draw *causal* conclusions from passively observed data using observational causal inference methods. This makes it possible to answer salient causal questions, such as *"Does smoking cause cancer?"*, without performing experiments, such as RCTs (see Section 2.2.4), which are often infeasible due to ethical or practical constraints. This challenge of answering causal questions without the ability to perform experiments also abounds in the software testing context albeit under a different guise.

As shown in Chapter 3, for certain classes of software such as computational models, metamorphic testing has the potential to be prohibitively expensive. In such circumstances, it is often the case that only a handful of metamorphic tests can be executed and only limited coverage can be achieved. As a consequence, we encounter what is essentially the same fundamental challenge: we want to answer a causal question (test some metamorphic relation) but cannot perform the necessary (quasi-) experiment to do so. In the metamorphic testing setting, the limiting factor is generally the cost involved in executing large numbers of metamorphic tests (as demonstrated in Chapter 3); in conventional settings of causal inference, such as epidemiology, the limiting factor is usually an ethical concern.

A question that naturally emerges from this shared problem is whether the same observational causal inference methods used to overcome this challenge in the epidemiological context can also be employed to the same effect in a testing context. This leads us to the following high-level question:

Can we similarly employ causal inference techniques to emulate metamorphic test outcomes using observational data?

Aside from the potential to leverage observational approaches to causal inference, metamorphic testing could also benefit from the extensive body of graphical causal inference methods (see Section 2.2.6). Most notably, causal DAGs (Definition 2.2) have emerged as a powerful tool for capturing domain expertise in a lightweight format that possesses a number of desirable properties. In particular, causal DAGs satisfy the causal Markov condition (Definition 2.3), meaning their structure implies a number of causal and (conditional) independence relations. In conventional causal inference settings, this result has been exploited to systematise the process of identification (i.e. identifying biasing variables).

In essence, this means that causal DAGs can assist domain experts in identifying causal and conditional independence relationships, and designing experimental or statistical procedures capable of testing them. Again, given the similarly causal objectives of metamorphic testing and causal inference, this leads us to the following high-level question:

Can we leverage graphical causal inference techniques to help identify metamorphic properties and tests?

Motivated by these high-level questions, in the remainder of this chapter, we explore how causal inference could, in theory, be applied to metamorphic testing. To achieve this, we extend the STF proposed by Staats et al. [1] to incorporate causal components that are necessary to facilitate the application of causal inference methods to causality-focused software testing problems, such as metamorphic testing.

4.2 Introducing Causality to the Software Testing Framework

In this section, we begin to consider what a *causal* testing framework would look like. To this end, we begin by briefly re-visiting the holistic software testing framework proposed by Staats et al. [1] before considering how we can modify and extend this framework to incorporate the additional attributes necessary for causal inference, such as a causal DAG. Using the resulting conceptual framework, in Section 4.3, we then express metamorphic testing as a problem of causal inference.

4.2.1 Revisiting the Staats et al. Software Testing Framework

As discussed in Section 2.1.1, Staats et al. [1] defined a theoretical testing framework that focuses on the interrelationship between four essential components of any software testing approach: the program-under-test P, the specification S, a set of tests T, and a set of test oracles O. In addition, the authors define a pair of predicates that capture the idealised objectives of testing: $corr \subseteq P \times S$ and $corr_t \subseteq P \times S \times T$. The former, $corr$, defines a program P as correct if it matches its specification S. The latter, $corr_t$, denotes the more granular assumption that a program P is correct with respect to a particular test $t \in T$ if, upon execution against t, P matches S.

The STF was motivated by the lack of research into the interrelationship between specifications, programs, tests and, in particular, test oracles. Prior to the publication of this work, software testing research had predominantly focused on the relationship between programs and tests in isolation, often leaving the role of the test oracle implicit and subsequently reducing the generalisability and interpretability of any findings. With this motivation in mind, the STF takes a holistic view of the interrelationship between programs, tests, and oracles, and emphasises the importance of considering their collective role in designing and evaluating testing techniques. One of the key suggestions made by the authors is to *state and defend all relevant assumptions* [1].

In the following section, we extend this framework with the goal of introducing the components necessary to conduct causal inference in a software testing setting. In addition to facilitating the application of causal inference to software testing problems, we also aim to continue the theme of making assumptions transparent that is pioneered in the STF by Staats et al [1]. This desire for transparent assumptions is also motivated by work on graphical causal inference, where researchers such as Judea Pearl and Miguel Hernán have advocated the use of graphical models to make causal assumptions abundantly clear and contestable. This motivation is best captured by Hernán's adage: *"Draw your assumptions before your conclusions"* [136].

4.2.2 Causal Testing Framework

We now begin to construct the CTF: an extension to the STF introduced by Staats et al. [1], with the aim of facilitating the application of causal inference techniques to software testing problems, while making testing and causal assumptions transparent and contestable.

To this end, we begin by providing a formal definition for the PUT, $\mathcal{P}$, before introducing the notion of *causal testing*[1] - a term we introduce to encompass testing activities that concern the causal effect of some intervention (see Definition 4.3). We then provide a definition for the specific form of test, oracle, and specification used in the CTF: *causal test cases*, *causal test oracles*, and *causal specifications*, respectively. In addition, we introduce data to the framework as its own first-class entity. This addition is motivated by the need for a technique that does not depend on potentially costly model executions and the importance placed on the data in the field of causal inference.

Program-Under-Test

For the scope of this thesis, we are primarily concerned with computational modelling software, which we have introduced and described in great detail in Chapter 3. However, where possible, we attempt to keep the CTF and its definitions general in case it can be useful elsewhere. To this end, we provide a general definition of the PUT, $\mathcal{P}$, which focuses on black-box systems. That is, a system for which we assume only knowledge of its input and outputs (i.e. its interface).

Definition 4.1 (*Program-Under-Test*)

In the context of the CTF, we consider the program-under-test (PUT) $\mathcal{P}$ to be a black-box system. We assume that $\mathcal{P}$ accepts a vector of inputs $\mathbf{I} = \langle I_1, I_2, \dots I_m \rangle$ and produces a vector of outputs $\mathbf{O} = \langle O_1, O_2, \dots O_n \rangle$, where each input and output is drawn from some appropriate domain of values $\text{dom}(I_x)$ and $\text{dom}(O_x)$, respectively. We refer to the input and output vectors as the interface of $\mathcal{P}$.

Example 4.1 (*Program-Under-Test (Covasim)*)

As an example PUT, let us consider Covasim (the ABM previously examined in Section 3.2) to be a black-box system whose behaviour can be configured by a vector of inputs $\mathbf{I}$ to produce a vector of output $\mathbf{O}$. For brevity, we now define $\mathbf{I}$ and $\mathbf{O}$ using a subset of the numerous inputs and outputs of Covasim.

Consider the following five inputs and their domains:

- The length of the simulation, `days`, whose domain is $\mathbb{Z}_{\geq 0}$ (i.e. the positive integers including zero)
- The initial population size, `pop_size`, whose domain is $\mathbb{Z}_{\geq 0}$
- The initial infectious population size, `pop_infected`, whose domain is $\mathbb{Z}_{\geq 0}$
- The location, `location`, whose domain is the set of 156 distinct locations available in

[1]This is distinct from the concept of "Causal Testing" used by Johnson et al. [103] for explaining the root cause of failures that was discussed in Section 2.3.2.

Covasim (e.g. UK and USA)

- The vaccine, `vaccine`, whose domain is the set of vaccines available in Covasim, such as Pfizer, as well as the option of no vaccine (None)

Moreover, consider the following two outputs and their domains:

- The cumulative number of infections at the end of the simulation, `c_infections`, whose domain is $\mathbb{Z}_{\geq 0}$
- The cumulative number of vaccinated individuals at the end of the simulation `c_vaccinated`, whose domain is $\mathbb{Z}_{\geq 0}$
- The cumulative number of vaccine doses given to individuals at the end of the simulation, `c_doses`, whose domain is $\mathbb{Z}_{\geq 0}$
- The maximum number of vaccine doses given to any individual across the simulation, `max_dose`, whose domain is $\mathbb{Z}_{\geq 0}$

Using these inputs and outputs, we can then the PUT as follows: $\mathcal{P} = (\mathbf{I}, \mathbf{O})$, where $\mathbf{I} = \langle \texttt{days}, \texttt{pop\_size}, \texttt{pop\_infected}, \texttt{location}, \texttt{vaccine} \rangle$ and $\mathbf{O} = \langle \texttt{c\_infections}, \texttt{c\_vaccinated}, \texttt{c\_doses}, \texttt{max\_dose} \rangle$.

Causal Testing and Causal Test Cases

Causal testing draws its main inspiration from causal inference, which focuses on the effects of *interventions* on *outcomes* to answer causal questions. In the software testing context, an intervention manipulates an input configuration in a way that is expected to *cause* a specific outcome to change. In keeping with causal inference nomenclature, we refer to the pre-intervention input configuration as a *control* and the post-intervention input configuration as a *treatment*. A *causal test case* then specifies the expected causal effect of this intervention.

For computational models that can simulate a diverse range of scenarios, such as the epidemiological models introduced in Chapter 3, it may be necessary to target our causal test cases towards specific modelling scenarios. For example, in Covasim (see Section 3.2) we may expect the intervention of introducing a vaccination policy to have drastically different effects in populations with different age distributions or contact patterns. A causal test for such a vaccination policy would therefore expect a different outcome depending on the population characteristics. We define a modelling scenario as a series of constraints placed over a subset of the PUT's input variables that characterise a scenario of interest. In many cases, however, the PUT may be sufficiently narrow in scope that it is sufficient to impose no constraints.

We provide formal definitions for an intervention, modelling scenario, and causal test case in Definitions 4.3 to 4.5, respectively, and give corresponding examples in Examples 4.3 to 4.5. To help construct these definitions, we also define the concept of an input realisation in Definition 4.2.

Definition 4.2 (*Input Realisation*)

An *input realisation* $\bar{\mathbf{I}}$ of a vector of inputs $\mathbf{I} = \langle I_1, I_2, \ldots, I_m \rangle$ is an assignment of each input $I_x \in \mathbf{I}$ to a value from its domain $\text{dom}(I_x)$, such that $\bar{\mathbf{I}} \in \text{dom}(I_1) \times \text{dom}(I_2) \times \ldots \times \text{dom}(I_m)$. We also refer to the input realisation as an input configuration interchangeably.

Example 4.2 (*Input Realisation - Covasim*)

Continuing with Covasim as our PUT (as defined in Example 4.1), we can define an input realisation of its inputs $\mathbf{I}$ as $\bar{\mathbf{I}} = \langle \texttt{days} := 200, \texttt{pop\_size} := 50000, \texttt{pop\_infected} := 1000, \texttt{location} :=$ UK, $\texttt{vaccine} :=$ Pfizer$\rangle$, where $I_x := v$ denotes assignment of the value v to variable I_x.

Definition 4.3 (*Intervention*)

An *intervention* $\Delta : \bar{\mathbf{I}} \rightarrow \bar{\mathbf{I}}$ is a function which replaces one input realisation with another. As short-hand, we use the notation $\bar{\mathbf{I}}[I_x := v]$ to denote a function that changes the value of variable I_x in the input realisation $\bar{\mathbf{I}}$ to the new value v.

Example 4.3 (*Intervention*)

Continuing with Covasim as our PUT from Example 4.1 and the input realisation $\bar{\mathbf{I}}$ defined in Example 4.2, suppose we want to define an intervention that replaces the Pfizer vaccine with an alternative vaccine, PfizerByAge, that prioritises the elderly (i.e. elderly individuals have a higher probability of being vaccinated). We can define such an intervention as follows: $\Delta_{\text{vac}}(\bar{\mathbf{I}}) = \bar{\mathbf{I}}[\texttt{vaccine} :=$ PfizerByAge$]$. This intervention yields the following input realisation: $\bar{\mathbf{I}}' = \langle \texttt{days} := 200, \texttt{pop\_size} := 50000, \texttt{pop\_infected} := 1000, \texttt{location} :=$ UK, $\texttt{vaccine} :=$ PfizerByAge$\rangle$.

Definition 4.4 (*Modelling Scenario*)

A *modelling scenario* $\mathcal{M}$ is a subset of input realisations that are expected to produce similar behaviour in the PUT. In practice, a modelling scenario is described by a set of constraints placed over a subset of the PUT's inputs, such as requiring a particular input be assigned a specific value.

Example 4.4 (*Modelling Scenario*)

Continuing with our running Covasim example, suppose we are interested in testing the effect of prioritising the elderly for vaccination (by applying Δ_{vac} defined in Example 4.3) on the four outputs defined in Example 4.1: `c_infections`, `c_vaccinated`, `c_doses`, `max_doses`. This means we are only interested in input configurations with one of two vaccines: Pfizer or PfizerByAge. Furthermore, assume we are only interested in simulations with a duration 50 days, an initial population size of 50000, an initial infectious population size of 100, and with the location set to the UK.

We can capture this behaviour of Covasim in a modelling scenario by defining the following set of constraints: {`days = 50`, `pop_size = 50000`, `pop_infected = 1000`, `location = UK`, `vaccine` $\in$ {Pfizer, PfizerByAge}}.

Definition 4.5 (*Causal Test Case*)

A causal test case $\mathcal{T}$ is a 4-tuple $(\mathcal{M}, \bar{\mathbf{I}}, \Delta, \mathcal{Y})$ that specifies the expected *causal effect* ($\mathcal{Y}$) of applying an intervention (Δ) to a particular input realisation ($\bar{\mathbf{I}}$) within the context of the modelling scenario ($\mathcal{M}$). Upon execution, a causal test case isolates and returns a measure of the *average* causal effect of the intervention, such as the ATE (Equation (2.2.6)) or RR (Equation (2.2.14)). Note that the expected causal effect $\mathcal{Y}$ is an informal expression of the change in output that is expected to be caused by executing $\mathcal{T}$.

Example 4.5 (*Causal Test Case*)

Continuing with our running Covasim example, suppose we are interested in constructing a causal test case to determine whether prioritising the elderly for vaccination causes a change in the maximum number of doses given to any agent. We can now define a causal test case using the modelling scenario $\mathcal{M}$, input realisation $\bar{\mathbf{I}}$, and intervention Δ defined in Examples 4.2 to 4.4. We then complete our causal test case by specifying the expected causal effect $\mathcal{Y}$. Since both versions of the vaccine (Pfizer and PfizerByAge) are configured to give a maximum of two doses per agent, we expect to observe *no change* in the maximum number of doses given to any agent.

Execution of a causal test case entails a three-step procedure. First, we check whether the specified input realisation satisfies the constraints of the modelling scenario. We explain the importance of this step later in Section 4.2.2. Second, we execute the PUT with both the pre- and post-intervention input realisations ($\bar{\mathbf{I}}$ and $\Delta(\bar{\mathbf{I}})$, respectively) to obtain two distinct vectors of outputs. Third, we then contrast the resulting output vectors to obtain a causal effect measure, such as the ATE (Equation (2.2.6)), to quantify the effect *caused* by the intervention.

Since the resulting causal effect is expressed as a population-level metric (explained previously in Section 2.2), causal test cases can be used in both deterministic and non-deterministic settings as the same metric can be calculated from one pair (deterministic systems where no repeats are necessary) or multiple pairs (non-deterministic systems where repeats are necessary) of pre- and post-intervention executions. As shorthand, we use the notation ATE_{Δ}^{Y} to denote the ATE of the intervention Δ on the output Y.

Causal Specification

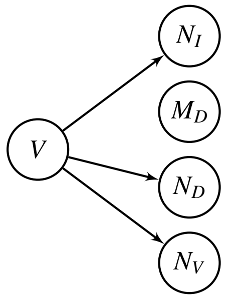
In order to construct the causal test cases from Definition 4.5, we require an explicit record of causal assumptions. To this end, we capture the expected causal behaviour of a computational modelling scenario using a causal DAG (Definition 2.2) that captures causality amongst a subset of the PUT's inputs and outputs. As a brief reminder, this is a DAG in which the direct edges between variables represent a causal relationship. Furthermore, as with the causal test case (Definition 4.5), we also consider the specification within the context of a particular modelling scenario. This means that it is only necessary to consider the causal relations amongst the variables that are key to the modelling scenario and not all variables in the computational model itself. We formally define a *causal specification* (Definition 4.6) as a pair comprising a modelling scenario ($\mathcal{M}$) and a causal DAG ($\mathcal{G}$). An example causal specification is given in Example 4.6

Definition 4.6 (*Causal Specification*)

A *causal specification* is a pair $\mathcal{S} = (\mathcal{M}, \mathcal{G})$ comprising a modelling scenario $\mathcal{M}$ and a causal DAG $\mathcal{G}$ capturing the causal relationships amongst the inputs and outputs of the PUT that are central to the modelling scenario.

Example 4.6 (*Causal Specification*)

Returning to our Covasim modelling scenario, we form our causal specification by constructing a causal DAG capturing the anticipated cause-effect relationships between the key variables involved in the scenario: V (vaccination policy), N_V (cumulative number of vaccinated individuals), N_D (cumulative number of doses administered), M_D (maximum doses administered to any given agent) and N_I (cumulative infections). Here, we add edges from V to all outputs apart from M_D. This reflects our assumption that changing the vaccination policy should have no effect on the maximum number of doses administered to any agent. Following this logic, we obtain the following causal DAG:



Collectively, this DAG and the modelling scenario outlined in Example 4.4 form our causal specification. Notice, however, that the fixed variables from the modelling scenario do not feature in the DAG because they are constants and therefore cannot be intervened upon (without violating the constraints of the scenario).

Causal Test Oracle

In the context of causal testing, the oracle - that is, the procedure used to determine whether the outcome of a causal test case is correct [15] - must speak the same causal language as the causal test case ($\mathcal{T}$) and causal specification ($\mathcal{S}$). This means a causal test oracle must ascertain the correctness of *causal effects* relative to a modelling scenario ($\mathcal{M}$). In its most basic form, a causal test oracle can be implemented as an assertion that captures the expected causal effect of some intervention, as defined by a causal test case.

Definition 4.7 (*Causal Test Oracle*)

A causal test oracle O is a procedure, such as an assertion, that determines whether the outcome of a causal test case $\mathcal{T}$ is correct or incorrect. This procedure checks whether the application of the intervention Δ to the input realisation $\bar{\mathbf{I}}$ has caused the expected causal effect $\mathcal{Y}$ in the context of modelling scenario $\mathcal{M}$.

We now provide three examples for causal test oracles: two of which we refer to as structural causal test oracles (Examples 4.7 and 4.8) and one of which we refer to as a functional causal test oracle (Example 4.9). The former are referred to as structural causal test oracles as they adjudicate the presence or absence of a causal effect (i.e. they establish causal structure). The latter is referred to as a functional causal test oracle as it anticipates a specific response to the intervention (i.e. the expected functional form of the causal relationship). In the following examples, we use the notation $\mathcal{P}^O\left(\bar{\mathbf{I}}\right)$ to denote an execution of a program, $\mathcal{P}$, with the input realisation $\bar{\mathbf{I}}$, which returns the output O.

Example 4.7 (ShouldCause)

Relative to a causal relationship $I \to O$, we can define the ShouldCause oracle as an assertion that checks whether some intervention, Δ, made to some input realisation $\bar{\mathbf{I}}$ of the input variable I causes *some* change to value of the output variable, O, when executed on program $\mathcal{P}$. That is, $\mathcal{P}^O\left(\Delta\left(\bar{\mathbf{I}}\right)\right) \neq \mathcal{P}^O\left(\bar{\mathbf{I}}\right)$ or, equivalently, $ATE^O_\Delta \neq 0$.

Example 4.8 (ShouldNotCause)

Relative to an independence relationship $I \perp\!\!\!\perp O$, we can define the ShouldNotCause oracle as an assertion that checks whether an arbitrary intervention, Δ, made to some input realisation $\bar{\mathbf{I}}$ of the input variable I causes precisely *no* change to the output, O, when executed on program $\mathcal{P}$. That is, $\mathcal{P}^O\left(\Delta\left(\bar{\mathbf{I}}\right)\right) = \mathcal{P}^O\left(\bar{\mathbf{I}}\right)$ or, equivalently, $ATE^O_\Delta = 0$.

Example 4.9 (*Functional Causal Test Oracle*)

Let us continue our vaccination example in Covasim. Suppose that we expect the introduction of the age-prioritised vaccination policy V (by applying the intervention Δ_V, as defined in Example 4.5, to input configuration $\bar{\mathbf{I}}$) to *increase* the number of cumulative infections, N_I. We can define the ShouldIncrease oracle as an assertion that checks whether Δ_V causes the cumulative infections N_I to increase as follows: $\mathcal{P}^{N_I}\left(\Delta_V\left(\bar{\mathbf{I}}\right)\right) > \mathcal{P}^{N_I}\left(\bar{\mathbf{I}}\right)$. Equivalently, we can express this as follows: $ATE^{N_I}_{\Delta_V} > 0$.

Data

The main deviation from the STF defined by Staats et al. [1] is the inclusion of data as its own first-class entity. While the various interrelationships defined within the STF implicitly rely on data - for example, that a test oracle should be considered in the context of its oracle data (the data used to determine whether some test should pass) - it is not included as a standalone component.

However, in the CTF, data is viewed with the same importance as tests, programs, oracles, and specifications. This is because we apply causal tests to data collected from a program, rather than the program itself. In addition, one of the main motivating factors behind applying causal inference to software testing is the potential to infer test outcomes from existing, passively observed data. To capture this key operational difference between the CTF and STF, we re-define the following predicates, *corr* and $corr_t$, that were originally introduced in the STF to define the notion of correctness with respect to a test and specification:

$$corr_t \subseteq T \times P \times S \tag{4.2.1}$$

$$corr \subseteq P \times S \tag{4.2.2}$$

In particular, Equation (4.2.1) captures the process a tester typically follows to test a program: they run the program P against test T and check whether it satisfies the specification S. Importantly, notice that this implies execution of the program itself; by contrast, causal testing operates on data collected from the PUT (D). To reflect this key difference, we replace P in Equations (4.2.1) and (4.2.2) with D:

$$corr_t \subseteq T \times D \times S \tag{4.2.3}$$

$$corr \subseteq D \times S \tag{4.2.4}$$

We now define two distinct forms of data corresponding to the two approaches to causal inference outlined in Sections 2.2.4 and 2.2.5: experimental and observational approaches.

Where data is collected by executing the PUT with the express purpose of performing a causal test case, we refer to the data as *interventional*. This essentially means that we perform the causal test directly, producing data that isolates the causal effect of interest by design in a similar way to an RCT.

Alternatively, where existing data from previous executions of the PUT are collected after the fact, we refer to the data as *observational*. Since this form of data has not necessarily been collected under the carefully controlled conditions needed to isolate the causal effect of interest, it may contain biases that must be identified and adjusted for using causal inference techniques.

For the purposes of inferring the outcome of a causal test case, we further classify data as being either *relevant* or *irrelevant* depending on whether it satisfies the constraints outlined in the modelling scenario of the causal test case (Definition 4.5). This means that only data that falls within the defined scope of the modelling scenario can be used to make inferences, leading to a trade-off between the specificity of the modelling scenario (and therefore underlying causal question) and the amount of data that can be considered valid.

We refer to this trade-off as the *inferential power* of the data, such that data has high inferential power if it is both relevant and abundant enough to support conclusive inferences for the causal test case in question. Although not formally defined nor quantified, this can be thought of as an informal type of data adequacy criterion.

Definition 4.8 (*Data*)

Data $\mathcal{D}$ is any artefact that can be collected from executions of the PUT, $\mathcal{P}$.

We say that data is *relevant*, relative to a modelling scenario $\mathcal{M}$, if it satisfies the constraints outlined by the modelling scenario. Otherwise, it is referred to as *irrelevant* and cannot be used for inference.

Moreover, relative to a causal test case $\mathcal{T}$, we classify data as *interventional* if it is the result of executing $\mathcal{T}$ directly. By contrast, if the data is collected from previous executions of $\mathcal{P}$, we say it is *observational*.

Example 4.10 (*Data*)

Consider our Covasim vaccination example. Suppose a team of epidemiologists have conducted numerous simulations to investigate the effect of different vaccination strategies in several locations, including the UK, France, and Japan. For each location, several different vaccination strategies have been tested using the Pfizer vaccine, such as prioritising the elderly, essential workers, and no one. These simulations have all been conducted using a simulation length of 50 days, 1000 initially infectious individuals, and a population size of 50000. The results of these simulations have been recorded and combined into a single CSV file in which each row represents an execution with a column for each input and output of interest. In this example, this CSV file forms our data, $\mathcal{D}$.

For the purpose of the causal test case outlined in Example 4.5 (i.e. to establish the effect of the prioritising the elderly in the UK, specifically), only the rows in which `location = UK` and `vaccine` $\in$ {Pfizer, PfizerByAge} are considered relevant. Moreover, since the data has not been generated for the precise goal of the causal test (multiple different interventions have been applied in multiple different locations), it is considered to be observational.

4.2.3 Comparison Between CTF and STF

Having introduced the components of the CTF, we now draw a direct comparison to the framework introduced by Staats et al. [1], which is also summarised in Table 4.1 below. Here, not only do we focus on the importance of these components to testing matters, but also how they help to satisfy the key identifiability conditions of causal inference (see Section 2.2.3).

Program Unlike the STF, which imposes no restriction on the nature of the program, the CTF is specialised to computational modelling software. We have, however, kept our definitions general where possible in case the CTF could be useful elsewhere.

Specification The first core difference arises in the specification. Since the CTF is designed with causal testing in mind - that is, testing activities that focus on the effect of a prescribed intervention - our specifications are causal in nature and reflect the anticipated cause-effect relationships in the PUT. This is in contrast to the idealised concept of a specification in the STF that perfectly captures the true, expected behaviour of the software.

In addition, as alluded to in Section 4.1, we are interested in exploring how graphical causal inference techniques can be used in the context of metamorphic testing. This requires a causal DAG (Definition 2.2), which is also provided by the causal specification.

This also plays an important function in satisfying the (conditional) exchangeability assumption (Equation (2.2.4)). Informally, this condition requires that the compared treatment and control populations are comparable in terms of all variables that may introduce bias to the causal effect of interest. This means

Component	Software Testing Framework	Causal Testing Framework
Programs	The program-under-test.	The computational modelling software-under-test.
Specifications	An abstract, ideal notion of correctness.	A causal specification, which captures the expected cause-effect relationships between the key variables involved in a specific modelling scenario, using a causal DAG.
Tests	Derived from the specification with the aim of finding executions that disagree with the specification.	A causal test case, which aims to establish the presence, absence, or functional form of the causal relationships outlined in the causal specification, in terms of the program's response to a defined intervention.
Oracles	A procedure that checks whether a test executed on a program produces correct or incorrect behaviour.	A procedure applied to program data that determines whether the program responds correctly to the intervention carried out by a causal test case. Such an oracle always measures correctness in terms of the causal effect.
Data	—	Any artefact produced by executing the program.

Table 4.1: A summary of the differences between the core components of the Software Testing Framework (STF) outlined by Staats et al. [1] and the Causal Testing Framework (CTF) proposed in this chapter.

that, where the goal is to infer testing outcomes from observational data, any biasing variables must be identified and adjusted for. Since this property cannot be verified in general, it is considered best practice to make your reasoning transparent (i.e. why a particular set of variables should or should not be adjusted for). To this end, causal DAGs provide a graphical justification for the design of any procedure used to estimate a causal effect and thus help to satisfy the exchangeability condition.

Tests The second difference lies in the tests. In the CTF, our tests are referred to as causal test cases and concern the causal effect of some intervention within the context of a defined modelling scenario. This decision is partially driven by our key takeaways from reviewing existing applications of causal inference to software engineering and testing problems (see Section 2.3.6). Consequently, our definition of a causal test case narrows the range of possible testing activities to those that concern the effect of an intervention (we refer to such activities as being causality-focused).

Causal tests also play an important role in communicating important information that is needed to satisfy the consistency assumption (Equation (2.2.10)). Informally, this requires that the intervention is well-defined and applied to the intended population and ensures that the estimated causal effects correspond to the right causal question. To this end, causal test cases communicate both the intervention and modelling scenario, the latter of which essentially defines the target population (the intended recipients of the intervention). Therefore, the causal test case contains the necessary information to remove ambiguity surrounding the intention of the test and interpretation of its results.

Oracles Given the causal focus of the specification and tests within the CTF, it follows that the test oracles must also be causal in nature. This means that the oracle data - that is, the data which the oracle uses to determine whether a test should pass or fail when executed on the program - must be some measure of causal effect. Informally, this means that the oracle must observe the outcome of a causal test case i.e. the effect of some prescribed change to the program's input configuration within a particular modelling scenario.

While the causal test oracle does not contribute to the identifiability conditions directly, it is similarly motivated to make any assumptions that may influence the interpretation of the results transparent. In particular, we make a key distinction between the expected outcome expressed by a causal test case ($\mathcal{Y}$) and the causal test oracle ($\mathcal{O}$): the former is a statement of the expected test outcome while the latter is the actual procedure used to check whether the anticipated outcome holds. We make this distinction to avoid situations where two testers may implement the procedure to ascertain the validity of a given causal test case in different ways, potentially leading to different test outcomes.

Data In addition to narrowing the scope of the programs, specifications, tests, and oracles, the CTF also includes data as a distinct entity. This is perhaps the most noteworthy change as it reflects a key difference between conventional software testing methods and causal testing - namely, that causal testing is applied to data collected from the PUT rather than the PUT itself. This decision is motivated by two key observations.

First, data plays a central role in causal inference. It is the conditions under which data is collected that ultimately determines which tools and techniques are suitable for making valid causal inferences. For example, in an RCT, the identifiability conditions are generally satisfied by design, meaning no further work is ordinarily required to reach causal status. However, where data is observational in nature, a great deal of additional work is needed to identify and correct various forms of bias. This problem also occurs in our software testing context, where a causal test may be executed directly to collect data under controlled conditions that isolate the causal effect, or it may be collected passively and thus require additional work to mitigate bias. For this reason, it is important to make the data itself (and particularly the nature of its collection) transparent.

Second, as mentioned in Section 2.2, causal inference offers the potential to predict causal effects from existing, passively observed and biased test data. Realised in a metamorphic testing context, this advantage could offer the potential to predict metamorphic test outcomes in computational modelling software using existing data, helping to address one of the key barriers outlined in Chapter 3.

In addition, given the information available in the modelling scenario and intervention of a causal test case, the CTF provides the necessary information to consider positivity violations. Informally, this assumption requires that we have sufficient coverage of the treatment and confounders in the data to estimate the causal effect of interest. This assumption can, in theory, be verified by breaking the data into all possible treatment-confounder combinations and flagging any empty strata. The aforementioned information can therefore be used to guide this process.

Having formally defined the CTF, in the following section, we give an example of how metamorphic testing — a fundamentally causal testing activity — can be translated to a causal inference problem.

4.3 Metamorphic Testing Using Causal Inference

We have developed our motivation for employing causal inference to alleviate the challenges associated with testing computational modelling software, and subsequently defined our theoretical CTF. Now we can formulate metamorphic testing as a causal inference problem. To demonstrate this process, we return to our Spanish Influenza EBM motivating example from Chapter 3.

After expressing metamorphic testing as a causal inference problem, we then discuss the prospective theoretical advantages offered by a causal inference-led approach to these problems. We relate these advantages back to the challenging properties of computational models outlined in Chapter 3 to highlight the suitability of causal inference techniques for testing such software.

This section sets the scene for the remaining chapters of this thesis, which, broadly speaking, aim to investigate how causal inference methods can improve the applicability of metamorphic testing to computational modelling software.

4.3.1 Framing Metamorphic Testing as a Causal Inference Problem

Let us consider MR5 from Table 3.5 that concerns the effect of decreasing the duration of death, γ_D, on the cumulative number of infected individuals, T_I, and deceased individuals, D, at the end of the simulation:

$$\begin{aligned} \text{Metamorphic relationship: } & \gamma'_D = n\gamma_D \text{ where } n < 1 \\ \text{Expected outcome: } & T_I \text{ and } D \text{ should decrease} \end{aligned} \tag{4.3.1}$$

Causal Specification

To provide our causal specification, we need to define a modelling scenario (Definition 4.4) and a causal DAG (Definition 2.2). In the paper that originally defined the metamorphic relations for the Spanish Influenza model [5], the authors provide a series of initial parameter settings to provide a fixed source test case and, for each input that is the focus of a metamorphic relation, a valid range of follow-up values. For MR5, we can capture these requirements in the following modelling scenario:

$$\{\sigma = 3, \alpha = 0.01, 0 < \gamma_D < 1, \gamma_R = 2.5, \tau = 0.15, \mu = 4, N = 103267000, I_0 = 1000, \beta = 0.6\} \tag{4.3.2}$$

Notice that we relax the constraint on γ_D to incorporate all valid interventions that reduce γ_D by a factor of $n \sim U[0, 1]$ as per Equation (4.3.4).

As explained in Section 3.3, some of the metamorphic relations defined in the original paper did not hold for all values of inputs that fall within the valid domain specified by the metamorphic relation. For example, for MR2 (concerning the effect of increasing the mortality probability, α, on the number of recoveries, R, and deaths, D), values of the input α exceeding 0.2 delayed the pandemic to such an extent that MR2 fails. Despite this critical change in behaviour, MR2 simply states that α should be increased by a factor $n > 1$. To address this issue, more precise input boundaries should be provided. We can achieve this within the CTF using a modelling scenario that places more restrictive constraints over the range of values for α:

$$\{\sigma = 3, 0.01 < \alpha < 0.2, \gamma_D = 1, \gamma_R = 2.5, \tau = 0.15, \mu = 4, N = 103267000, I_0 = 1000, \beta = 0.6\} \tag{4.3.3}$$

To complete our causal specification, we capture the expected cause-effect relationships between the inputs and outputs involved in MR5 in the causal DAG shown in Figure 4.1. This causal DAG is straightforward in the sense that it simply communicates the assumptions that (i) all inputs affect both outputs, and (ii) that β is determined by values of τ and α as per Table 3.2. Optionally, we could also simplify the DAG to remove all nodes representing variables which are constrained to a fixed value by the modelling scenario, since these variables are then reduced to constants and cannot be intervened upon to evoke a causal effect. This simplified DAG is shown in Figure 4.2.

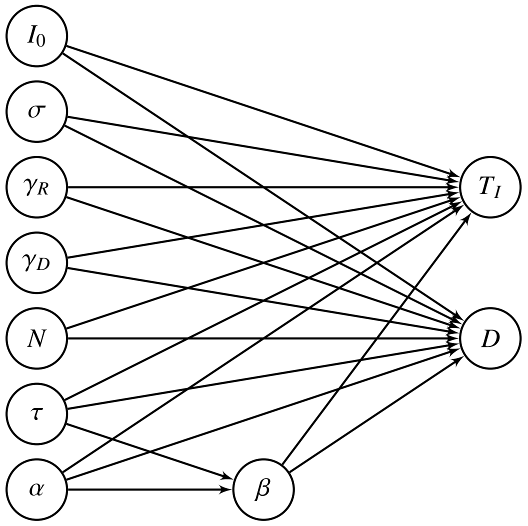


Figure 4.1: A causal DAG capturing the expected cause-effect relationships between the inputs and outputs involved in MR5 (Table 3.5 and Equation (4.3.1)) for the Spanish Influenza model, which was introduced as a motivating example in Section 3.1.

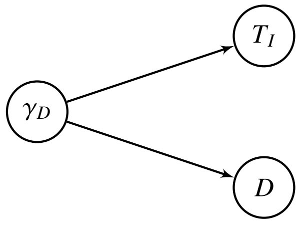


Figure 4.2: A simplified causal DAG for the Spanish Influenza model Section 3.1 that contains only non-constant inputs and the outputs of interest as per the corresponding modelling scenario (Equation (4.3.2)).

When performing metamorphic testing in the conventional sense (i.e. by executing the PUT under controlled conditions), a causal DAG offers no advantage. However, in Chapter 5 and Chapter 6, we show how causal DAGs such as Figure 4.1 can be leveraged to overcome the limitations surrounding metamorphic testing outlined in Chapter 3. More specifically, in Chapter 5, we show how metamorphic relations can be systematically derived from causal DAGs representing the key cause-effect behaviour in software systems. In Chapter 6, we explore how causal DAGs can be paired with graphical causal inference methods to predict metamorphic test outcomes from existing execution data, making it possible to predict how the model would respond to interventions that cannot be performed due to issues of practicality (e.g. cost).

Causal Test Case

The metamorphic relationship in Equation (4.3.1) states that, when γ_D is decreased by a factor $n < 1$, T_I and D should decrease. Intuitively, this makes sense as upon decreasing the duration of death, the individuals who contract a fatal infection (i.e. will eventually die) will die sooner and therefore be removed from the simulation sooner too. This reduces the amount of time these individuals are infectious for and therefore lowers their propensity to spread the virus, resulting in fewer cumulative infections and, in turn, fewer deaths.

Metamorphic relations, such as the one in Equation (4.3.1), essentially communicate the expected *causal effect* of some *intervention*. That is, the change in some output(s) caused by a prescribed change to some input variables. This enables us to extract two of the key components needed to construct a causal test case (Definition 4.5) from the metamorphic relation shown in Equation (4.3.1)): the intervention (Δ) and the expected causal effect ($\mathcal{Y}$).

In the case of MR5, the intervention is a decrease to the input parameter γ_D. We can capture this as the following intervention, where $\bar{\mathbf{I}}$ denotes the full input configuration:

$$\Delta_{MR5}(\bar{\mathbf{I}}) = \bar{\mathbf{I}}\,[\gamma_D := n \times \gamma_D] : n \sim U\,(0,1)^{2} \tag{4.3.4}$$

We then specify the expected causal effect ($\mathcal{Y}$) of this intervention, which is a decrease in both T_I and D.

Since we have already defined our modelling scenario ($\mathcal{M}$) all that remains is to specify our control input configuration $\bar{\mathbf{I}}$. This corresponds to the default parameter settings listed in Table 3.2:

$$\langle \sigma := 3, \alpha := 0.01, \gamma_D := 1, \gamma_R := 2.5, \tau := 0.15, \mu := 4, N := 103267000, I_0 := 1000, \beta := 0.6 \rangle \tag{4.3.5}$$

Causal Test Oracle

As noted in our causal test case, we expect the intervention Δ_{MR5} to cause a *decrease* in T_I and D. It is then the job of the causal test oracle to determine whether this expected outcome has been observed, given the outcomes of the causal test case. In this case, this can be implemented as an assertion that checks whether both ATEs are negative. We can express this formally as follows:

$$\left(ATE^{T_I}_{\Delta_{MR5}} < 0\right) \wedge \left(ATE^{D}_{\Delta_{MR5}} < 0\right) \tag{4.3.6}$$

Data

In order to test MR5 in the CTF, we require execution data from the Spanish Influenza model that conforms to the constraints outlined in the modelling scenario (Equation (4.3.2)). Given the lightweight nature and low execution cost of this computational model (as described in Section 3.1), it is reasonable to directly execute the causal test case (i.e. collect interventional data).

For a deterministic system such as the Spanish Influenza EBM, the causal test case can then be executed as follows. First, we execute the PUT with both pre- and post-intervention input configurations to obtain

[2] The notation $n \sim U\,(0,1)$ denotes that the value of n is drawn at random from a uniform distribution between 0 and 1.

a control and treatment outcome. The resulting execution data is considered interventional since it was collected for the express purpose of the causal test case. Consequently, we can calculate the ATE by simply contrasting the collected control and treatment outcomes. Formally, we can express this process as follows, where $\mathcal{P}^O(\bar{\mathbf{I}})$ denotes an execution of the Spanish Influenza EBM with input configuration $\bar{\mathbf{I}}$ that returns the value of the variable O at the end of the simulation:

$$\begin{aligned} ATE^{T_I}_{\Delta_{MR5}} &= \mathcal{P}^{T_I}\left(\Delta_{MR5}\left(\bar{\mathbf{I}}\right)\right) - \mathcal{P}^{T_I}\left(\bar{\mathbf{I}}\right) \\ ATE^{D}_{\Delta_{MR5}} &= \mathcal{P}^{D}\left(\Delta_{MR5}\left(\bar{\mathbf{I}}\right)\right) - \mathcal{P}^{D}\left(\bar{\mathbf{I}}\right) \end{aligned} \tag{4.3.7}$$

This process can then be repeated for different versions of the same intervention (i.e. for different sized decreases in γ_D). By virtue of directly executing the causal test case, all data is relevant (i.e. no data falls outside of the constraints specified by the modelling scenario).

Summary

Having defined the causal specification, causal test case, data, and causal test oracle, we now possess all components of the CTF and have therefore defined metamorphic testing as a problem of causal inference. We can now apply our causal test oracle to the collected data to measure whether the defined intervention, $\Delta_{MR5}\left(\bar{\mathbf{I}}\right)$, has had the anticipated causal effect on the specified outputs T^I and D (i.e. caused a decrease). In this situation, due to the lack of practical restrictions placed on the EBM, our data was gathered by executing the model directly. This means we have essentially conducted metamorphic testing in the conventional way: we have executed a source and follow-up test case which isolate the causal effects of interest by design. This symmetry reflects the inherent causal nature of metamorphic testing.

Later, in Chapter 6, we provide a reference implementation of this framework and show how it can be applied to both interventional and observational data collected from real-world computational models. In the latter case, this causal inference-led approach demonstrates the viability of predicting metamorphic test outcomes from existing data; a more cost effective alternative to conventional metamorphic testing in situations where practical issues abound, such as those outlined in Chapter 3. However, this is only one advantage offered by causal inference. In the following sub-section, we highlight a number of potential advantages that causal inference (and thus the CTF) could offer in a software testing context.

4.4 Theoretical Advantages of the CTF

The formulation of causal testing activities, such as metamorphic testing, as causal inference problems offers a number of theoretical advantages:

1. Transportability: First, by simply framing testing activities within the CTF, we are essentially translating the testing task to a problem of causal inference. And since the components of the CTF have a direct correspondence to the key ingredients for causal inference (i.e. an intervention, an outcome of interest, a causal DAG, and some data), we can directly apply existing causal inference techniques to these problems. In this way, the CTF provides an interface between software testing and causal inference that has much untapped potential for leveraging a wide range of established causal inference techniques that could be of benefit to a myriad of causality-focused testing activities.

2. Cost effective test executions: Where a testing problem can be captured within the CTF, observational causal inference techniques offer the potential to address these testing problems using existing, potentially biased data. This divorces the testing task from costly software executions, offering a feasible alternative to testing for classes of software that are ordinarily considered to be 'untestable' (see Section 2.1.4). Hence, in the same way that observational causal inference has been used to answer challenging causal questions that cannot be answered experimentally (e.g. *Does smoking cause lung cancer?*), it could be employed to test causal relationships where practical constraints prevent conventional testing approaches from doing so.

3. Universal and testable properties: Everything that is dynamic (in the sense that it can *intervene* or *effect change* in some way), including software systems, have a causal structure that can be captured using a causal DAG. Causal DAGs are supported by an extensive mathematical framework that, amongst other things, enables a series of causal and independence properties to be derived from a causal DAG based on its structure. Hence, if we can express the expected behaviour of some software component as a causal DAG, we can utilise these techniques to identify a series of testable properties. Since these properties are causal in nature, they can be viewed as a kind of metamorphic relation as they specify how the modelled system should respond to particular changes. Furthermore, since causality is universal, these properties are too. This means causal inference has the potential to offer a means for efficiently deriving metamorphic relations from a causal DAG. This has the potential to shift the task of identifying metamorphic relations to one of drawing a causal DAG.

4. Transparency: Staats et al. strongly encourage software testing research to communicate and justify all assumptions surrounding the key components of the STF. Causal inference research shares a similar philosophy: the validity of causal inferences depend on the assumed causal structure, data, and experimental or statistical procedure used for estimation. Therefore, it is essential that these factors are made transparent and contestable. The sentiment of both fields is particularly important in the context of computational modelling software, where there is a long standing and well documented separation between testers and model developers. One of the main problems is that test oracles are not readily communicated; the correctness of a computational model is often eye-balled by a domain expert with no further justification. By providing an interface between software testing and causal inference techniques, the CTF has the potential to bridge this gap between software developers and domain experts. This is because it makes both the testing and causal assumptions transparent and contestable. In particular, the causal assumptions are expressed in a causal DAG, which is a lightweight and intuitive graphical model that is easy to understand for domain experts and software testers alike. Furthermore, the CTF enables domain experts to test their software in familiar terms that are agnostic of the implementation.

4.5 Concluding Remarks

This chapter has motivated the potential for causal inference to provide a solution to the problem statement outlined at the end of Chapter 3, which highlights two key barriers that currently hinder the application of metamorphic testing to computational modelling software:

1. The difficult and time-consuming process of constructing metamorphic relations.

2. The execution cost resulting from the significant computational overhead associated with compu-

tational models.

At the start of this chapter, we discussed how causal inference has been employed in fields such as epidemiology to answer challenging yet important causal questions that, due to ethical or practical issues, cannot be answered experimentally. We then drew a parallel with similar problems that hinder metamorphic testing of computational modelling software, giving rise to the following questions that ultimately motivate **RQ2.1** and **RQ2.2** from Section 1.1:

Can we similarly employ causal inference techniques to emulate metamorphic test outcomes using observational data?

Can we leverage graphical causal inference techniques to help identify metamorphic properties and tests?

With these questions in mind, in Section 4.2, we adapted the STF originally proposed by Staats et al. [1] to form the CTF, which facilitates the application of causal inference techniques to causality-focused testing activities. The CTF was designed with causality and the nuances of computational modelling software in mind, as well as the need to clearly communicate causal and testing assumptions. To this end, the CTF comprises refined definitions for specifications, tests, and oracles that are causal in nature. The CTF also includes data as its own entity since data plays a central role in causal inference and should therefore be communicated with the same transparency as other components.

After defining this framework, in Section 4.3, we gave an example of how, given its fundamentally causal objective, metamorphic testing can be expressed within the CTF. We then outlined several potential advantages of formulating causal testing activities as problems of causal inference, including the possibility to leverage graphical causal inference techniques to identify causal and independence relations, and observational causal inference techniques to predict causal test outcomes from existing test data.

To summarise, in this chapter, we have:

- Discussed how similar problems to those faced in the testing of computational models abound in fields such as epidemiology, albeit under a different guise, and how causal inference has been used to overcome these problems.
- Outlined how the challenges presented by the properties of computational models and the limitations of metamorphic testing may be mitigated using causal inference.
- Introduced a theoretical framework – the CTF – that frames causality-led testing activities, such as metamorphic testing, as causal inference problems.
- Framed metamorphic testing as a causal inference problem within the CTF and outlined the potential advantages of this formulation.

Following this chapter, we progress to Part II of this thesis, which focuses on exploring how the aforementioned theoretical advantages can be realised in practice to increase the applicability of metamorphic testing to computational models. In particular, we focus on how causal inference can mitigate the two key barriers identified in our problem statement (Section 3.4): the difficult and time-consuming process of constructing metamorphic relations (Chapter 5), and the execution cost resulting from the significant computational overhead associated with computational models (Chapter 6).

Part II

Incorporating Causal Inference into Metamorphic Testing

5 | Identification of Metamorphic Relations from Causal Graphs

With the help of two epidemiological models as motivating examples, in Chapter 3, we demonstrated some of the potential limitations of metamorphic testing in the computational modelling domain, ultimately leading us to the problem statement outlined in Section 3.4. Given our problem statement, in Chapter 4, we then examined causal inference as a potential solution and outlined the conceptual CTF, which enabled us to frame metamorphic testing as a causal inference problem.

In this chapter, we focus on the first challenge outlined in our problem statement: the task of identifying metamorphic relations. As explained previously in Section 2.1.5, although metamorphic testing helps to alleviate the test oracle problem, it does not remove it entirely; rather, it shifts the challenge from identifying the 'correct' outputs corresponding to some inputs, to identifying suitable metamorphic relations. These relationships are usually domain-specific in nature and manually identified, making it a non-trivial task that calls upon significant domain expertise.

With this challenge in mind, in this chapter, we start to build upon our conceptual CTF with the aim of leveraging causal inference techniques to assist with the identification of metamorphic relations. Here, we exploit the causal nature of metamorphic testing (as previously discussed in Section 4.1) and leverage graphical causal inference techniques to propose a systematic and domain agnostic model-based approach for the identification of metamorphic relations.

This chapter is based on publication **P2**: *Metamorphic Testing with Causal Graphs* (ICST, 2023).

5.1 Introduction

As noted previously in Section 2.1.5, a key barrier to the widespread application of metamorphic testing is the difficulty of formulating metamorphic relationships [43, 44]. Although several advances into semi-automatic metamorphic relation construction have been made [44], existing approaches have significant limitations. Techniques that reverse-engineer metamorphic relations from executions [51] risk reverse-engineering faulty relations if the underlying code is faulty. Other approaches are restricted to specific domains [49].

Ultimately, the construction of metamorphic relations is a difficult task for which there is no technique that is both generalisable and reliable. Instead, the burden of this task typically falls to an experienced tester or domain expert who must rely upon their ingenuity and intuition. A more in-depth discussion of metamorphic testing approaches is given in Section 2.1.5.

The approach proposed in this chapter is based on the observation that metamorphic testing is an inherently *causal* activity: a transformation is applied to an input in order to establish the causal effect on an output. When viewed from this perspective, and as discussed in Chapter 4, metamorphic testing has an extremely similar objective to that of causal inference (Section 2.2), namely: to measure and establish a causal effect. To the best of our knowledge, there are no existing efforts that utilise causal inference methods to systematically identify metamorphic relations.

In this chapter we propose a causal, model-based approach to the generation of metamorphic relations and test cases. Our approach is centred around causal DAGs [59] (Definition 2.2), a well-established and relatively lightweight graphical model that has been used extensively to model causal relationships in a wide range of disciplines, such as epidemiology. In particular, we show how causal DAGs can be used to model the expected causal relationships between variables in a software system, and how the resulting structure can be used in turn to automatically generate metamorphic relations and their test cases. Through a series of controlled experiments and illustrative case studies, we then evaluate the ability of our approach to detect a specific class of bugs that modify the causal structure of the PUT and may appear in any class of software.

Overall, in this chapter, we make the following contributions:

- We present an approach to metamorphic testing that uses causal graphs to automatically generate metamorphic relations with their source and follow-up test cases.
- We perform a controlled experiment to analyse the ability of our approach to detect bugs that affect the causal structure of the PUT under increasingly challenging conditions.
- We apply our technique to software from the Defects4J framework with known bugs that affect the causal structure.

The remainder of this chapter is structured as follows. In Section 5.2, we introduce the concepts of total and direct effect that are necessary for this chapter. In Section 5.3, we define the notions of *structural* and *functional* causal bugs and give rudimentary examples of both. Then, in Section 5.4, we describe the process for systematically identifying metamorphic relations given a causal DAG representing the PUT. In Section 5.5, we present an empirical evaluation of our technique before discussing the results in Section 5.6 and demonstrating its application to two real-world case studies in Section 5.7. We then discuss the main findings of the study and limitations of the approach in Section 5.8. Finally, we conclude the chapter in Section 5.9.

5.2 Total and Direct Effects

Given a treatment X and an outcome Y, there are two main forms of causal relationships that can be measured: the *total* and *direct* effect [137]. We can characterise these with the help of a causal DAG (see Definition 2.2).

The total effect of X on Y includes the effect of so-called *mediators*; variables that lie on a directed path between X and Y [58]. For example, consider the causal DAG shown in Figure 5.1, in which Z is a mediator of the effect of X on Y. To measure the total effect (TE) of changing the value of X from x to

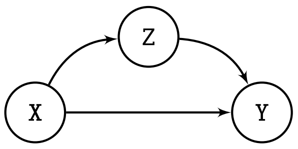


Figure 5.1: An example causal DAG for the causal effect of `X` on `Y` mediated by `Z`

x', we simply measure the outcome for both values of the treatment, X, and observe the resulting change in Y, while blocking any back-door paths. In this example, this corresponds to the following formula:

$$\text{TE} = Y^{X=x'} - Y^{X=x} \tag{5.2.1}$$

Notice the correspondence with the back-door criterion: any adjustment set that satisfies the back-door criterion is sufficient to measure the total effect.

Second, we can measure the direct effect of X on Y, which excludes the effect of any mediators. To measure the direct effect (DE), we additionally block the flow of information along all mediating paths. This is achieved by adjusting for the value of at least one mediator, Z, on this path. This corresponds to the following formula:

$$\text{DE} = Y^{X=x',Z=z} - Y^{X=x,Z=z} \tag{5.2.2}$$

The intuition here is to block all paths between X and Y apart from the path $X \to Y$ itself [137], which enables us to measure only the direct effect of X on Y. For example, a simple method to achieve this is to adjust for all parents of Y apart from X itself. Note that, in the presence of any confounding, it would also be necessary to block any open back-door paths.

5.3 Causal Bugs

A bug or fault is some static defect that exists in software [4] that may *cause* the software to behave in an erroneous way. In this chapter, we are particularly concerned with bugs that affect causal relationships between the inputs and outputs of a program. To aid this discussion, we introduce two types of causal bugs: *structural* and *functional* causal bugs. While we are particularly concerned with the former in this chapter, we also introduce the latter for completeness.

5.3.1 Structural Causal Bugs

Implicitly, all programs have some underlying causal structure. If we change the value assigned to particular program elements, such as variable values or the inputs to a method, we expect this to change the way in which the software behaves. For example, in a simple statement: `y = 2*x + 10`, the variable `x` has a causal effect on `y`. That is, if we change the value of `x`, `y` will change in response. Collectively, such causal relationships make up the causal structure of a program[1].

In some cases, a bug may inadvertently manipulate the causal structure of the PUT. For example, a developer may reference the wrong variable or call the wrong method. As a consequence, one of two

[1]Aside from the implicit notion that all programs have a causal structure, there are several precedents that have used various forms of program dependence graph as a graphical model of a program's causal structure in the field of fault localisation, as discussed in Sections 2.3.1 and 2.3.4.

events may occur. First, the developer may introduce an unexpected side effect: a variable that should not have an effect now does. This could introduce an entirely new and unexpected causal relationship, or erroneously modify the structure of an existing and expected one. Second, the developer may remove an anticipated effect: a variable that should have had an effect no longer does. Notice that, in both cases, the causal structure of the program is changed.

We refer to such a bug as a *structural causal bug*. To illustrate this concept clearly, consider the ReLU function, a form of activation function used in deep learning that converts all negative inputs to zero:

$$\text{ReLU}(x) = \max(0, x) \tag{5.3.1}$$

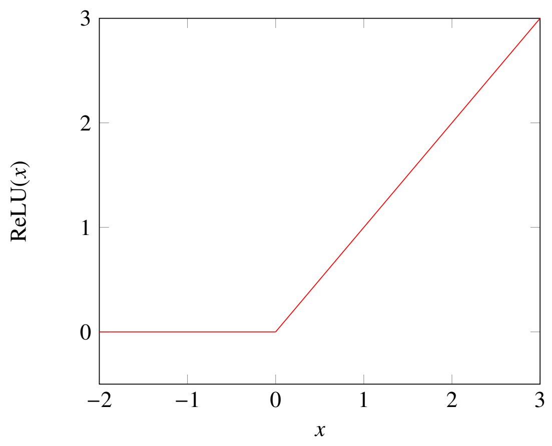


Figure 5.2: Functional form of ReLU.

We use the ReLU function from Figure 5.2 as an illustrative example as it enables us to clearly demonstrate what a causal relationship $X \to Y$ represents and, in turn, the concept of a structural causal bug. Most importantly, notice that it is *possible* to select two values of X for ReLU that produce *different* values of Y (where X and Y represent the input and output of ReLU, respectively). For example, any pair comprising a negative and non-negative value will yield 0 and some positive integer, respectively. Here, we say that X causes Y (denoted $X \to Y$) as we can perform *some* intervention (see Definition 4.3) on X that evokes a response from Y. We emphasise "some" here as $X \to Y$ does not mean that *any* change to X will cause Y to change, merely that there exists at least one possible change to X that will cause Y to change. This is an important distinction as it demonstrates the relatively weak assumption that $X \to Y$ conveys.

As an example to consolidate this point, consider that if we select any two negative inputs for ReLU, both will yield the same output: 0. Therefore, we could define an intervention which changes X from -1 to -2, for example, and this would cause no change to Y. Despite this, we still say that X causes Y since we can perform other interventions on X (e.g. change its value from 1 to 2) that would cause Y to change. The key point being: $X \to Y$ means *some* changes to X affect Y but not necessarily all.

Now, consider the following buggy version of ReLU in which $X \not\to Y$:

$$\text{ReLU}(x) = \max(0, 0) \tag{5.3.2}$$

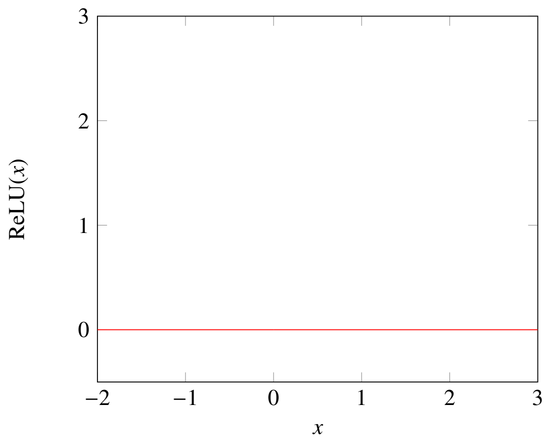


Figure 5.3: An example structural causal bug in ReLU.

Here, regardless of the value of input X, ReLU *always* returns 0 as shown in Figure 5.3. Therefore, there exists no change to the input X that can cause the value of its output Y to change. In this case, the causal structure of the program is manipulated as a causal relationship that should exist ($X \to Y$) does not. Critically, the absence of a causal relationship — independence (denoted as $X \perp\!\!\!\perp Y$) — conveys the stronger assumption that precisely no intervention on X can evoke a response in Y, and vice versa.

In a secondary analysis of Sobreia et al.'s study [138] of patches to the Defects4J framework [17], it was found that 57.7% of bug fixes in Defects4J were labelled with a repair action that implied a change in causal structure [139]. For example, repairs tagged with the addition or removal of program variables imply that a new causal relationship may have been introduced or an erroneous one removed. Similarly, repairs labelled as wrong variable reference imply that a variable in an existing relationship has been replaced by another. This form of bug can easily be introduced by selecting the wrong variable name suggested by an IDE, for instance. Hence, it appears that structural causal bugs are prevalent, motivating the need for their detection.

Throughout this chapter, we focus on developing a method that is capable of assisting to developers in systematically identifying metamorphic relations that are capable of detecting these structural causal bugs. Later, in Section 5.7, we study two real-world examples of structural causal bugs and demonstrate how the approach we propose in Section 5.4 can detect these.

5.3.2 Functional Causal Bugs

While this chapter focuses on structural causal bugs, it is worth noting a complementary form of bug that may also impact causality in software systems. Suppose we have the ground truth causal structure of a program as test oracle data. That is, a source of information that can be used to ascertain the correctness of our program. This would enable us to determine whether or not the expected cause-effect relationships between inputs and outputs hold in the PUT. However, it would tell us nothing about *how* the output Y should respond to particular changes in input X, where there is a causal relationship $X \to Y$. Therefore, even when the causal structure of a program is as expected, there may still be a problem with the precise functional form of the relationships between inputs and outputs. We refer to such bugs as *functional* *causal bugs*.

As an example, let us consider another buggy version of ReLU:

$$\text{ReLU}(x) = \max\left(0, \frac{x}{2}\right) \tag{5.3.3}$$

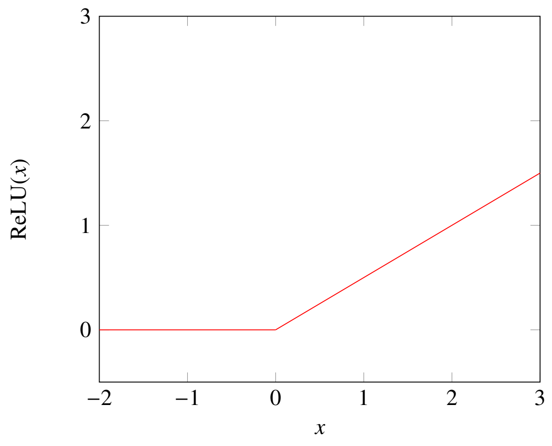


Figure 5.4: An example functional causal bug in ReLU.

Here, the causal structure is correct. As with Equation (5.3.1), there exists some change in X that will evoke a response in Y e.g. changing X from -1 to 1 will change Y from 0 to 0.5. However, as per the intended functional form of ReLU given in Equation (5.3.1), in this buggy version of ReLU, changes to X that evoke a response in Y will change its value by half as much as expected, as shown in Figure 5.4. For example, changing X from -1 to 1 should change Y from 0 to 1, but actually changes it from 0 to 0.5. Therefore, although the causal structure is correct, the precise functional form is not. This is an example of a functional causal bug and would require a much finer-grained test oracle to detect. We leave this to future work but introduce it here for completeness.

5.4 Causal Metamorphic Testing

This chapter is based on the observation that there is a strong relationship between causal DAGs and the type of behaviours that are expressed in metamorphic relations. Both are fundamentally based on the notion of a "causal effect". Implicitly, metamorphic relations (Definition 2.1) perform an intervention (Definition 4.3) that transforms a given input (from the source value to the follow-up value) in such a way that it *causes* a change in output that can be anticipated. The notion of an edge in a causal graph (Definition 2.2) similarly incorporates the fact that a causal relationship between two variables occurs if an intervention on one may *cause* the other to change. Specifically, if there exists an edge $X \to Y$, this carries the assumption that there exists some intervention over X that will evoke a response (i.e. cause a change to) from Y.

In this chapter, we explore the relationship between the two areas of metamorphic testing (see Section 2.1.5) and graph-based causal reasoning (see Section 2.2.6). This raises the prospect of taking advantage of the significant amount of research on graph-based causal reasoning to identify and test potential metamorphic relations.

5.4.1 Causal DAGs as a Software Model

In order to leverage the causal nature of metamorphic testing, we start by modelling the PUT in the form of a causal DAG (Definition 2.2). For this we consider a simple abstraction that is subtly different to our previous definition of the PUT (Definition 4.1), in that it generalises beyond black-box systems:

- There are "inputs", which correspond to parameters or environment variables that can affect the functional behaviour of the program.
- There are "observable variables", which correspond to observable results of a computation by the PUT.

Based on this abstraction, for the purpose of this chapter, we introduce the notion of a **Causal Software Graph** (CSG) which combines the definitions of a causal DAG (Definition 2.2) and the PUT (Definition 4.1). This is formally defined in Definition 5.1.

Definition 5.1 (*Causal Software Graph*)

A Causal Software Graph (CSG) is an extension of a causal DAG (see Definition 2.2) defined as a tuple (V, E, I, O). Here, V and E correspond to the nodes and edges of the causal DAG, and the sets I and O represent "inputs" and "observable variables", respectively, such that $(I \cup O) = V$ and $I \cap O = \emptyset$. Additionally, we have $\forall (x, y) \in E.\, x \in O \implies y \notin I$, i.e. observable variables cannot cause inputs. This must hold true in all software systems[2] since inputs must be provided before any computation can be done.

Where software inputs are known to be independent, we may also assume $\forall (x, y) \in E.\, \neg(x \in I \wedge y \in I)$. That is, there are no input to input causal relationships. This is not a requirement of our technique, but it does reduce the size of the test suite as we can ignore input-input tests.

When creating a CSG, it makes sense to start from the interface of the PUT. That is, its inputs and outputs. From this we can determine the set I: the input parameters and environment variables that we can control as part of a test setup. We can also identify the set of outputs that we wish to check, and use this as a basis for O. It may also be helpful to capture additional key states of computation by including some internal state variables in O. This typically requires instrumentation of the PUT. We discuss this in Section 5.4.2.

The edges between the variables represent our expectation of how variables should affect each other. As with most model-based testing techniques [140], the resulting model captures the developer's (or domain expert's) *expectation* of how the system should behave. Because of this, the CSG can only be as accurate as the developer's understanding of the PUT, and may end up being an approximate abstraction of the system. We investigate the degree to which our techniques can cope with model misspecification in Section 5.5.

[2]Iterating behaviour where the result of one iteration feeds into the next can be represented by either unrolling a fixed number of iterations or making the graph more abstract to consider the inputs and outputs of the iteration.

```
Input: A CSG $(V, E, I, O)$, a number $N$ of metamorphic tests to be generated per causal relationship.
Output: A set of metamorphic tests $MT$, where each test is structured as
        $(source, followUp, output, relation)$.
1  begin
2      $MT \leftarrow \emptyset$;
3      for $x \in V$ do
           /* For every pair of variables that could feasibly form a causal edge
              */
4          for $y \in (V \setminus (I \cup \{x\}))$ do
5              $adjustmentSet \leftarrow identify(x, y, E)$;
6              for $i = 0; i < N; i = i + 1$ do
                   // Create a source input for variables in I
7                  $source \leftarrow generateRandom(I, adjustmentSet)$;
                   /* Create a follow-up input, where variable $x$ is transformed to
                      a different value                                          */
8                  $followUp \leftarrow transform(source, x, adjustmentSet)$;
9                  if $(x, y) \in E$ then
                       /* If (x, y) is a causal edge, changing X should cause Y to
                          change                                                 */
10                     $MT \leftarrow MT \cup \{(source, followUp, y, \neq)\}$;
11                 else
                       /* If (x, y) is not a causal edge, changing X should not
                          affect Y                                               */
12                     $MT \leftarrow MT \cup \{(source, followUp, y, =)\}$;
13     return $MT$;
```

Algorithm 1: Generating Metamorphic Relations and Tests from a causal DAG, assuming $\forall(x, y) \in E.\neg(x \in I \wedge y \in I)$.

5.4.2 Generating Metamorphic Relations and Tests from a CSG

Algorithm 1 shows how a CSG can be used to identify a series of metamorphic relations and generate corresponding metamorphic tests. The basic principle is to generate a metamorphic test for every edge in the CSG to check that a change to the variable at the source of the edge (whilst holding variables in the adjustment set constant) has a causal effect on the target variable. Similarly, if there is no edge between two variables, a metamorphic test is generated to check that changing the source variable does not cause a change to the target variable. In this way, the metamorphic tests correspond to a pair of primitive metamorphic relations that check for ‘should cause’ and ‘should not cause’ relationships.

In this sub-section, we discuss the following three main steps of Algorithm 1 below: identifying the adjustment set, test case generation, and metamorphic relation construction. Following this, we briefly discuss the process of executing the resulting metamorphic tests and highlight a considerable practical barrier: hidden variables.

Identifying the Adjustment Set

The approach begins by iterating through every pair of nodes (x, y), where y is not an input (lines 3 and 4). Here we assume $\forall(x, y) \in E. \neg(x \in I \land y \in I)$ from Definition 5.1, although we could simply iterate over all node pairs in $V \times V$ to account for systems with input-input dependencies. For each pair of nodes (x, y), we proceed to identify an adjustment set (line 5) that isolates the causal effect of interest. We intentionally abstract the details of this process from the algorithm as there are various different adjustment sets capable of deriving different types of causal estimates. For example, by using the back-door criterion (Definition 2.5), a tester can obtain an adjustment set that measures the *total effect* (Equation (5.2.1)). Alternatively, by additionally adjusting for mediators, a tester can measure the *direct effect* (Equation (5.2.2)).

In this chapter, we use the *direct effect* (Equation (5.2.2)) as our causal effect measure on the basis that it provides a more granular test oracle than the total effect would. This is because it captures the expected effect of an *individual* causal relationship, rather than *multiple* related causal relationships. For example, if we have a causal graph comprising the edges $x \to y$ and $x \to z \to y$, the total effect would include the effects of three closely related causal relationships: $x \to y$, $x \to z$, and $z \to y$. Now, to illustrate why this may be problematic, suppose that a structural causal bug exists that removes the intended direct effect $x \to y$. Using the total effect, the absence of $x \to y$ may be obscured by the mediating path $x \to z \to y$. That is, while x does not change y directly, it still changes y indirectly through its effect on z. Alternatively, if we test these relationships separately as direct effects (i.e. $x \to y$, $x \to z$, and $z \to y$), we could detect that $x \to z$ and $z \to y$ but $x \not\to y$.

To isolate the direct effect, we require a set of variables that blocks all paths between x and y, apart from the direct path $x \to y$ itself (if this edge exists). Critically, this means that any mediating paths $x \to z \to y$ are blocked by including z (the mediator) in the adjustment set. In addition, to satisfy conditional exchangeability (Equation (2.2.4)), we must block all back-door paths. One simple and computationally inexpensive method that achieves this is to adjust for all parents of y except for x itself and all parents of x (if any exist). Formally, we can express this as the following formula, where $Pa(v)$ denotes the immediate parents of v:

$$adjustmentSet \leftarrow Pa(x) \cup Pa(y) \setminus \{x\} \tag{5.4.1}$$

It is worth noting, however, that in practice, there may be alternative preferable adjustment sets for any given effect measure. For example, it would be desirable to avoid adjustment sets including variables that are inaccessible by default as this would necessitate additional effort, such as instrumentation of the PUT. For this reason, we keep Algorithm 1 general in the sense that it does not prescribe a specific effect measure nor identification procedure; rather, these can be modified to the needs of the user, provided the conditions of the identification criterion corresponding to the selected effect measure are met e.g. the back-door criterion (Definition 2.5) to identify the total effect (Equation (5.2.1)).

Test Case Generation

After identifying our adjustment set, we then set about constructing our metamorphic relation. As a reminder, we define a metamorphic relation as a triple $MR = (x_s, T, R)$, comprising the source test case, the transformation function used to obtain the follow-up test case, and the property to test, respectively (Definition 2.1).

We start by generating the source test case, (x_s), using the *generateRandom* function (line 7), which samples a random value for each variable from I and the adjustment set. The follow-up test case, (x_f), is then obtained by applying the *transform* function, (T), (line 8) to the source test case, which changes the value of the treatment variable x to a fresh random value, while holding all variables in the adjustment set constant. This is repeated to generate the desired number of metamorphic test cases (N). Again, the precise method for generating test inputs could be substituted with any preferred method of test case generation.

Metamorphic Relation Construction

Finally, guided by the structure of the causal DAG, we complete our metamorphic relation by specifying the property to test, R. Specifically, if $(x, y) \in E$, we check for inequality between the source and follow-up outputs (line 10) since the causal edge means that particular changes to x should cause an observable change in the outcome y. We refer to metamorphic relations of this form as `ShouldCause`. This form of relation passes provided *at least one* test causes a change in y. Conversely, it fails only if *no* test causes a change in y.

By contrast, if $(x, y) \notin E$, we check for equality instead (line 12) as, if there is no causal effect between x and y, changes to x should never cause changes to y. We refer to metamorphic relations of this form as `ShouldNotCause` relations. The criterion for passing this form of relation is more strict: *all* tests cases must cause no change to y. Conversely, this metamorphic relation fails if *at least one* test causes a change in y.

Note that a passing `ShouldNotCause` relation does not prove the absence of a causal relationship. Rather, it demonstrates that none of the tested transformations of x caused a change to y. To draw the stronger conclusion that x does not cause y would require exhaustive testing of the input space of x which would be infeasible for most non-trivial systems. In the same vein, if a `ShouldCause` relationship fails, it does not mean there is no causal relationship, but that it has not been exercised by the tested transformations. The key point here is that without testing all possible transformations, of which there may be an intractable number, we cannot prove the absence of causality[3]. However, this can be partially mitigated by constructing large test suites that cover a greater proportion of the input space.

Metamorphic Test Execution

To execute our metamorphic test cases, we need to be able to intervene on all variables that appear as an input of interest or in the adjustment set. While this is trivial for input variables (i.e. those in I), we may also need to access and modify the values of variables in O. For example, the PUT may be an implementation of an algorithm that manipulates the state of its input by applying multiple functions sequentially. By default, we would only have access to the input and output of this algorithm, corresponding to its initial and final state. However, we may be interested in tracing whether its state is modified in the expected way upon each step of the algorithm. We refer to such variables that are not accessible by default as *hidden*. The means by which to access these variables (and the extent to which this can be automated) depends on the nature of the PUT. If, for example, the PUT is a Java class and variables in O correspond to class-level variables, it is possible to use bytecode instrumentation frameworks such as Javassist (we do this for our second case study). In other cases, some manual instrumentation or testabil-

[3]This is essentially the same challenge as faced in all forms of testing: for most non-trivial systems, it is impractical or even impossible to prove the absence of bugs.

ity transformations [141] may be required or desirable (we use manual instrumentation for our first case study).

For some systems (e.g. when the system is a true “black box”), these approaches may simply be infeasible. In such cases, the inability to fully control and observe relevant variables presents a fundamental limitation in testability [142]. In the worst case, this may prevent the testing of causal effects involving hidden variables, and the CSG may have to be restricted to the subset of accessible variables.

However, depending on the topology of the CSG, it may also be possible to *estimate* causal effects involving hidden variables from existing, passively observed data instead [143, 144]. Alternatively, under particular conditions, so-called instrumental variable methods can be used to overcome the problem of unobserved confounding [145]. While we do not apply these techniques in this chapter, in Chapter 6, we show how ‘observational’ causal inference methods can be used to overcome similar testability limitations and ultimately reduce the cost associated with executing metamorphic tests.

Now that we have defined and explained the approach underpinning this chapter, in the following section, we consider the conditions under which the technique is capable of capturing any bug that would affect the causal structure of the PUT, before moving onto our experimental evaluation which investigates how the technique performs as these conditions are systematically deteriorated.

5.4.3 Correctness and Performance

Given the two following strong assumptions, our approach is guaranteed to catch *all* bugs that change the causal structure of the PUT.

- **A1** *No model misspecification*: If the CSG perfectly reflects the causal structure of the PUT, we can generate a metamorphic relation capable of exercising every causal and independence relationship.
- **A2** *Injective causal relationships*: If every causal relationship $X \to Y$ in the CSG is implemented in the PUT as an injective function (i.e. $f(x_s) = f(x_f) \implies x_s = x_f$), then, for a given metamorphic relation, all possible source and follow-up test case pairs, (x_s, x_f) where $x_s \neq x_f$, will exercise the causal or independence relationship.

Informal proof. Let F denote a software function and $G = (V, E, I, O)$ denote a CSG representing its causal structure. Using **A1** and the definition of a CSG (Definition 5.1), we know that for every pair of variables $(X, Y) \in V \times V$, if $(X, Y) \in E$, then there exists at least one statement in F where Y is assigned a value dependent on X. Conversely, if $(X, Y) \notin E$, there is no such statement in F. Thus, by iterating over the edges and non-edges of F, as outlined in Algorithm 1, we obtain a complete set of metamorphic relalations that cover all causal and independence relations in the PUT.

If **A2** holds, we can verify each metamorphic relation using an individual (x_s, x_f) pair. If F implements each causal relationship as an injective function $Y = f(X)$, we have $f(x_s) = f(x_f) \implies x_s = x_f$, and equivalently $x_s \neq x_f \implies f(x_s) \neq f(x_f)$. Thus, any pair satisfying $x_s \neq x_f$ is sufficient to test an $X \to Y$ relationship. For independence relations, if $x_s \neq x_f \wedge f(x_s) = f(x_f)$, we know that X cannot be a term in the function, so the independence relation holds. □

As with any model-based testing technique, the model is an abstraction of the PUT and, in real settings, the fidelity of this will vary. In the following section, we systematically relax **A1** and **A2** to reduce the fidelity of our CSGs, both in terms of their accuracy (**A1**) and the proportion of non-injective causal relations in the programs they represent (**A2**). While these assumptions are unlikely to hold in general, they provide a basis to quantify and control model fidelity and, ultimately, understand how this impacts the usefulness of the generated metamorphic relations and tests.

It is also worth noting that misspecification is not the only way **A1** can be undermined. As discussed in Section 5.4.2, the (lack of) testability of the PUT can play a role as well. Some of the variables that should be in the CSG may not feature because the corresponding variables in the PUT cannot be controlled or observed [142].

5.5 Evaluation

In Section 5.4, we provided an informal proof that our technique can detect all structural causal bugs in an injective program, given a CSG which perfectly reflects its causal relationships. However, these assumptions are unlikely to hold in general. In this section, we investigate how our technique performs when these assumptions break down. Our research questions are as follows:

RQ1 *How robust to model misspecification is causal metamorphic testing?*
As with all forms of specification, in real settings, the CSG is likely to contain imperfections. It is therefore important to understand how such imperfections impact the performance of causal metamorphic testing, particularly in terms of the validity of the generated metamorphic relations and ability to detect structural causal bugs.

RQ2 *How robust to increasing amounts of non-injective causal relationships is causal metamorphic testing?*
When causal relationships are not injective, we face the additional challenge of finding a fault-revealing intervention, as the causal effect is only exercised by certain configurations. This research question aims to understand the extent to which the presence of evasive, non-injective causal relationships impacts the ability to detect structural causal bugs, and how this relationship varies with different sized randomly generated test suites.

5.5.1 Experimental Setup

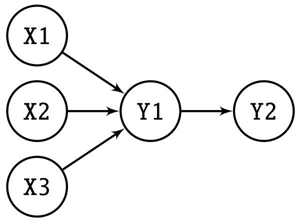
To answer **RQ1** and **RQ2**, we require software with a ground-truth CSG. To this end, we generated random CSGs and synthetic programs that reflect their causal structure, as in Figure 5.5. We then used mutation analysis to inject artificial bugs that either added or deleted a causal relationship in the implementation. We now describe the approach used for each of these steps in greater detail.

CSG Generation

We first generate a random undirected graph using the Erdős-Rényi model [146], and assign each node a numerical label. This model has two parameters, n and p_e, corresponding to the number of nodes and probability of including an arbitrary edge, respectively. We then convert the undirected edges to directed edges such that all edges point from nodes with a lower label to nodes with a higher label. Finally, we convert all endogenous nodes to inputs and all exogenous nodes to observable variables, and update the labels to denote inputs as X and observable variables as Y. This produces a CSG such as the one shown in Figure 5.5b. We then generate tests from the CSGs as described in Section 5.4.2, sampling parameter values uniformly at random from the range [-10, 10].

```python
def program(X1,X2,X3,Y1=None,Y2=None):
  if Y1 is None:
    if X1 + X3 <= 2:
      Y1 = 2*X1 + 3*X2 + 2
    else:
      Y1 = 2*X1 + 2
  if Y2 is None:
    Y2 = 3*Y1 + 2
  return {'Y1': Y1, 'Y2': Y2}
```

(a) An example synthetic program.



(b) Corresponding CSG.

Figure 5.5: A synthetic CSG and Python code reflecting the causal structure.

Program Generation

Our programs take the form of Python methods comprising a series of linear equations for each node in O. We generate the program corresponding to a CSG by iterating over the nodes of the graph in order and generating an equation for each that includes a linear combination of all its cause nodes and a randomly sampled constant. For example, for the CSG in Figure 5.5b, we could produce the equation `Y1 = 2*X1 + 3*X2 + -3*X3 + 2`. The method takes all nodes in I as mandatory parameters and all nodes in O as optional parameters. Each linear equation is then nested in an `if` statement that checks whether the effect node has been specified as an argument, in which case the argument overrides the equation. This allows us to intervene directly on any variable by passing in the appropriate argument.

Note that the linear equations comprising the synthetic programs are not, by default, injective. If we have the equation `Y = X1 + X2`, for example, we can select multiple pairs of values for `X1` and `X2` that cause `Y` to take on the same value. However, when performing Algorithm 1, we fix the value of all inputs and vary *only* the value of the input of interest. Consequently, supposing we are testing whether `X1` causes `Y`, the function is essentially reduced to the form `Y = X1 + C` where `C` is some value that remains constant between the source and follow-up executions. Contrary to its original form, in this equation, `X1` and `Y` share an injective relationship as every unique value of `X1` maps to a unique value of `Y`. Therefore, in the context of our experiment, we can treat these relationships as injective.

Furthermore, in order to examine how the performance of the technique copes with non-injective relationships, we add a "conditional behaviour" parameter, p_c. In essence, this controls how difficult the causal effects are to observe by introducing conditional causal relationships that are, by definition, non-injective. This parameter specifies the probability that a (randomly selected) node in O is conditional, meaning its assignment is nested in an if-then-else statement. To do this, we generate two linear equations — one for each branch — and a predicate is generated as a random inequality check that compares the sum of a random non-empty subset of the effect node's parents to a random value in the range [-10, 10]. The resulting conditional node only appears in one of the two branches and therefore can only have an effect when a particular condition (i.e. the predicate) is satisfied.

For example, consider the if-then-else statement surrounding the definition of `Y1` in Figure 5.5a. Because of the predicate `if X1 + X3 <= 2` and the absence of `X2` in the else branch, `X2` $\rightarrow$ `Y1` is not injective: we *must* cover the `if` branch to observe the effect of `X2` on `Y`. Therefore, the source or follow-up test case (or both) must satisfy the predicate `X1 + X3 <= 2`, increasing the challenge of observing the causal effect of `X2` on `Y1`. Otherwise, if neither satisfies the predicate, changes to the value of `X2` will have no observable impact on `Y1`: $x_s \neq x_f \not\Longrightarrow f(x_s) \neq f(x_f)$ (i.e. violates the injective property).

Mutation Analysis

We define two mutation operators to add and remove causal relationships in a given program, referred to as `AddCause` and `DeleteCause`, respectively. We implemented these operators in a fork of the Python mutation analysis framework, Cosmic Ray [147, 148].

The `AddCause` operator adds a causal relationship from a specified cause node to a specified effect node. Our implementation finds definitions of the effect node in the source code and adds the cause node to the existing linear equation. For example, `AddCause` could add `X2` as a causal effect to `Y1 = 2*X1 + 3` as follows: `Y1 = 2*X1 + 3 + X2`. Where the effect node is defined conditionally (i.e. a different linear equation per branch), it is sufficient to add a causal effect to *either* branch to introduce a new causal relationship. This is because an edge $X \to Y$ in a causal graph (Definition 2.2) means there exists *some* intervention on X that brings about a change in Y. For this reason, our mutation operator acts on each branch individually, producing two mutants.

Conversely, the `DeleteCause` operator removes all causal relationships from a cause node to an effect node. Our implementation finds definitions of the effect node that include the cause node and replaces the cause node with a randomly sampled numerical constant. For example, `DeleteCause` could remove the causal effect of `X2` on `Y1` in `Y1 = 3*X1 + 2*X2 + 1` as follows: `Y1 = 3*X1 + 2*5 + 1`. Furthermore, where the effect node is defined conditionally, `DeleteCause` acts on both branches and the predicate to ensure that the effect is entirely removed.

Based on the structure of the ground truth CSG, we automatically define a list of applicable mutation operators for each program. Specifically, we define a `DeleteCause` mutation operator for each edge in the CSG, and an `AddCause` mutation operator for each unconnected pair of nodes in the CSG to which an edge could be added to form a valid CSG (i.e. without cycles, outputs causing inputs, or inputs causing inputs).

5.5.2 Methodology

Each experiment has three parameters n, p_e, and p_c, pertaining to the number of nodes, probability of including an arbitrary edge, and probability of making a node conditional, respectively. We generate CSGs for every combination of parameters, $n = [10, 20, 30]$, $p_e = [0.25, 0.5, 0.75, 1]$, and $p_c = [0.25, 0.5, 0.75, 1]$. To mitigate the potential for bias introduced by the random CSG and SUT generation approaches, we repeat each configuration 30 times with different random seeds. This results in 1440 CSG-program pairs that vary in size, edge density, and conditional complexity.

RQ1: Robustness to misspecification

To introduce model misspecification, for each ground truth CSG, we produce four mutant CSGs in which each edge or absence thereof is inverted (deletion of an existing edge or addition of a new one) with probability $p_m = [0.25, 0.5, 0.75, 1]$. To quantify the extent of misspecification, we measure the **Structural Hamming Distance** (SHD) [149, 150]. Put simply, this metric measures the number of changes that need to be made to restore the misspecified CSG to the ground truth.

Following Algorithm 1, we generate our metamorphic relations and tests, before applying the mutation analysis approach described in Section 5.5.1. We then plot the relationship between mutation score and SHD to understand how the performance of the technique varies as the extent of misspecification increases.

RQ2: Robustness to conditional behaviour

As before, we perform mutation analysis on each of the generated programs. However, to answer this research question, we additionally report the McCabe cyclomatic complexity of the program, which quantifies the complexity of source code as a function of the number of edges, nodes, and connected components in its control flow graph [151][4]. We then study the relationship between the mutation score and the McCabe cyclomatic complexity to understand how the performance of the technique varies as the programs increase in complexity. Here we are using the McCabe complexity as a proxy for the extent to which the injective assumption ($A2$) is violated and, therefore, how difficult it is to exercise the causal relationship-under-test. For this research question, we use the strata of data with SHD = 0 i.e. data using only the ground truth DAG and therefore no misspecification.

5.6 Results

In this section, we present the results for our experiments designed to answer **RQ1** and **RQ2**. Although we controlled the size n of the generated CSGs to repeat our analysis with 10, 20, and 30 nodes, there were no apparent differences between the corresponding results that would have a baring on the outcome of the research questions. Therefore, we focus our discussion on the results for $n = 20$. Results for the other sizes can be found in our repository[5].

5.6.1 RQ1: Robustness to misspecification

The results in Figure 5.6 show that, as the CSG is increasingly misspecified (relaxing assumption **A1** from Section 5.4), the mutation score falls with the SHD. This suggests that, where the user cannot provide a perfect (or even particularly accurate) CSG, the proposed technique can still produce metamorphic relations and tests for the correctly specified attributes of the CSG that reveal structural causal bugs.

We observe that the mutation score drops more steeply for CSGs with higher values of p_e (i.e. for CSGs that have a denser edge structure). This can be explained in terms of the effects of the causal mutation operators on the adjustment sets of the generated metamorphic relations. When a spurious causal edge is added, the adjustment sets of metamorphic relations may then include additional variables, but are still sufficient to isolate the causal effect, meaning the test outcome is unaffected. By contrast, removing causal edges removes variables from the adjustment sets that are necessary to isolate the causal effect. This can lead to additional, correctly specified metamorphic relations failing on the *unmutated program* (making them ineligible to catch mutants) because they fail to adjust for all variables necessary to isolate the causal effect. This is consistent with the general consensus within causal inference that the absence of a causal edge is a stronger assumption than the presence of one [152].

Overall, our results lead us to the following answer to **RQ1**:

[4]Specifically, the McCabe cyclomatic complexity M of an individual method is calculated as: $M = E - N + 2$, where E and N represent the number of edges and nodes in the method's control flow graph. We use the McCabe Python package to achieve this: `https://github.com/PyCQA/mccabe`.

[5]`https://github.com/AndrewC19/causal_metamorphic_relation_generation/tree/main/full_results`

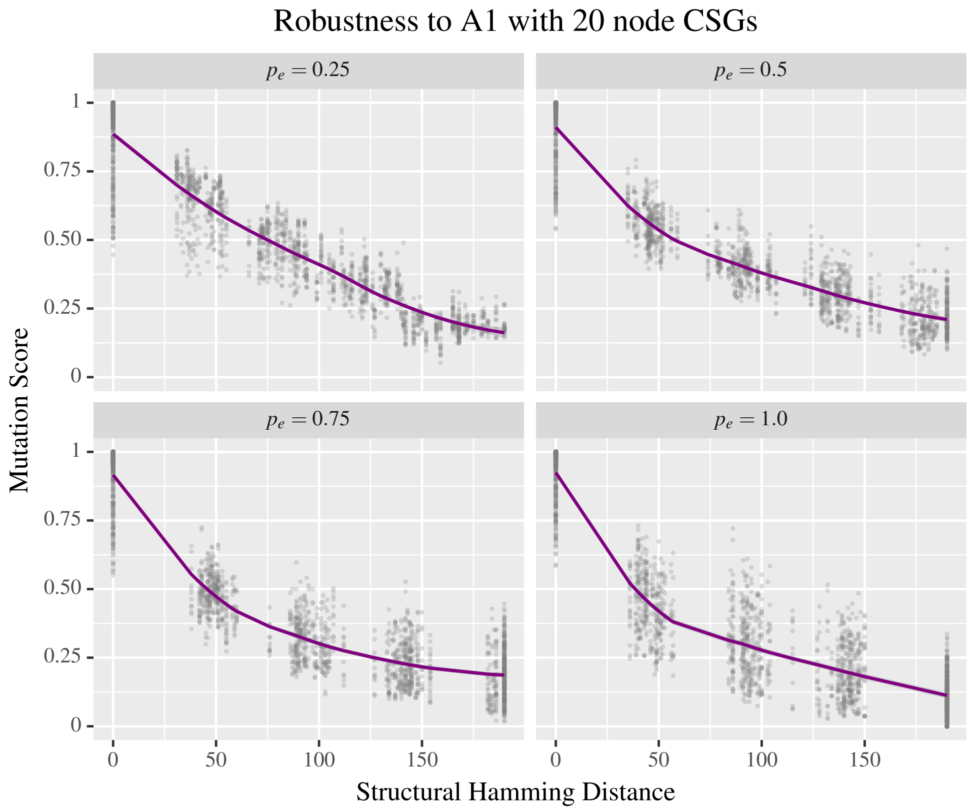


Figure 5.6: Results for **RQ1** demonstrating the relationship between structural hamming distance (SHD) and mutation score for CSGs with 20 nodes and different edge densities (p_e).

RQ1: Our results indicate that the proposed approach is robust to misspecifications in the underlying CSG. They also suggest that missing genuine edges has a greater adverse effect on fault detection capability than including spurious ones.

5.6.2 RQ2: Robustness to evasive causal relationships

The first row of the results in Figure 5.7 shows that, as the McCabe complexity increases and thus a greater proportion of causal relationships are made conditional (relaxing assumption **A2** from Section 5.4), the mutation score falls drastically when using just a single metamorphic test. Specifically, over all edge-densities, the mean mutation score decreases from almost 0.97 down to approximately 0.68 as the CSG increases in complexity. For larger test suites, however, this effect becomes negligible. For a test suite of size 5, the mean mutation score falls from 1 to 0.95. For a test suite of size 10 it falls from 1 to 0.98.

By comparing the trends for programs with different edge densities, we can see that the mutation score at the greatest McCabe complexity falls as the density of edges decreases. This indicates that, in the pres-

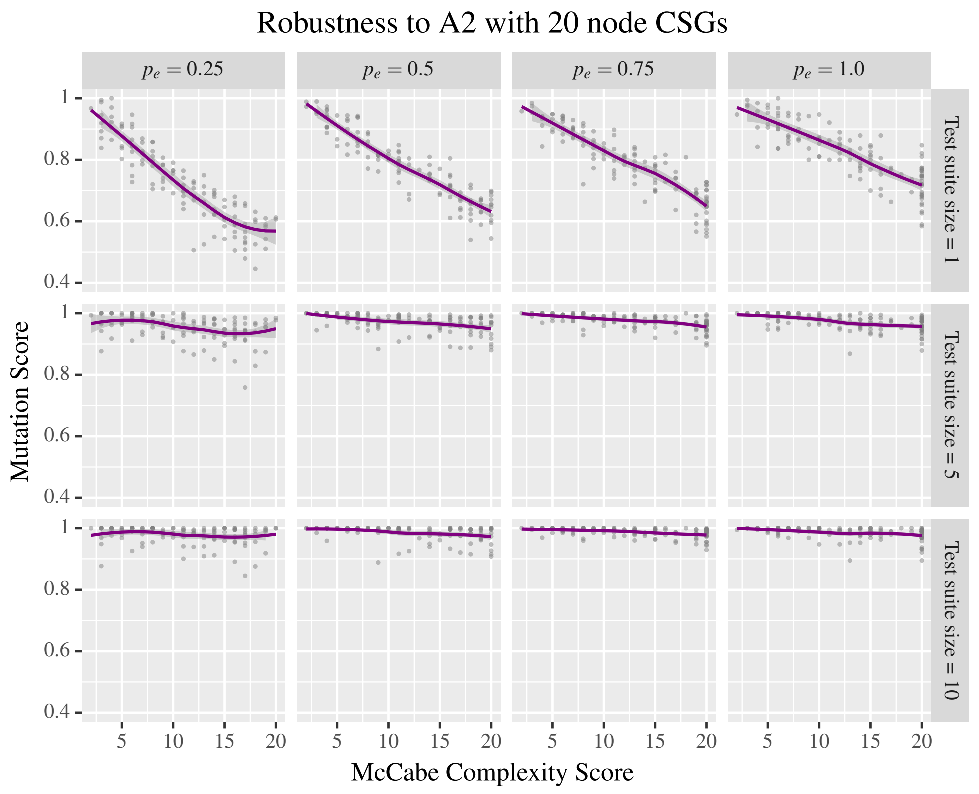


Figure 5.7: Results for **RQ2** demonstrating the relationship between McCabe complexity and mutation score for CSGs with 20 nodes and different edge densities (p_e).

ence of conditional behaviour, it may be more difficult to detect the addition of a spurious relationship than the absence of a genuine one. This finding follows logically from the nature of the mutation operators introduced in Section 5.5.1 - namely, that the `AddCause` operator introduces a cause to *one* of the conditional branches, whereas the `RemoveCause` operator must remove the cause from *both branches* and the *predicate itself*. Intuitively, the latter leaves a larger fingerprint that is easier to detect.

These findings indicate that faults residing in non-injective causal relationships (introduced here by conditional behaviour) can be notably harder to detect and, therefore, that the presence of conditional behaviour greatly increases the testing challenge associated with detecting structural causal bugs. However, the results also show how an improved test generation strategy can help to address this issue, with the larger randomly generated test suites achieving significantly better mutation scores. This indicates that the robustness of the proposed approach to conditional behaviour could likely be improved by pairing it with a more advanced test generation strategy.

Overall, based on the above findings, we answer **RQ2** as follows:

RQ2: Our results indicate that the ability of the proposed approach to detect structural causal bugs can be adversely affected by the presence of non-injective causal relationships. However, this issue may be mitigated by using more rigorous test suites for metamorphic testing.

5.6.3 Threats to validity

We now discuss a series of identified external and internal threats to validity concerning our experiment.

External threats to validity

The main threat to external validity is that we generated synthetic programs in a parameterised, controlled manner. This was necessary to produce programs with a known causal structure that could be configured to achieve the aims of the experiment (e.g. add and delete causal relationships, and introduce conditional causal behaviour). However, they are unlikely to be representative of realistic programs. This concern motivated our case studies in Section 5.7 where we apply our technique to two actual programs from the Defects4J repository.

Another threat is that we employed a random test generation strategy to explore how the robustness to non-injective (evasive) causal relationships varies with the size of a test suite. Using a more advanced coverage-driven approach would undoubtedly have achieved better results. However, the fact that our naïve approach can still achieve reasonable mutation scores demonstrates the promise of our technique.

Internal threats to validity

Our internal threats to validity are as follows. First, since we randomised the generation of the CSGs, programs, and mutation configurations, our results are subject to stochasticity, and are not guaranteed to provide a representative picture. We mitigated this threat by repeating each configuration 30 times as per the guidelines of Acruri et al. [153].

Secondly, the metrics we selected to measure the extent of misspecification (SHD) and non-injective behaviour (McCabe complexity) may not precisely resemble the target quantity. However, we chose these metrics because they are well known in their respective fields: SHD is commonly used to measure the accuracy of causal DAGs in the field of causal discovery[6][154], and McCabe complexity is a common metric for measuring software branching behaviour that indicates the complexity of a program.

It is also worth acknowledging that we did not compare our approach to a baseline or existing techniques such as [155, 45]. Since there are relatively few metamorphic relation generation techniques in the literature, there does not seem to be a commonly accepted baseline or set of comparison metrics. Therefore, it is unclear how the performance of our technique would compare to existing methods for generating metamorphic relation from the literature. We leave this to future work.

5.7 Case Studies

In Section 5.5, we evaluated our technique on synthetic programs reflecting the causal structure of randomly generated CSGs. Our aim was to investigate how the fault-finding capability of our technique

[6]This is a distinct field to causal inference that is interested in the inverse problem: to learn causality from data, rather than using causal assumptions to analyse data.

changes as assumptions **A1** and **A2** from Section 5.4 deteriorate. In this section, we apply our technique to detect structural causal bugs from two systems in the Defects4J repository [17]. The implementation of these case studies can be found in our repository[7].

5.7.1 Apache Commons Math

First, we focus on a bug in the `evaluation` method of the `Variance` class in version 2.2 of the Apache Commons Math library (Bug ID: `Math 41`). According to the documentation[8], this method calculates the weighted variance of a specified portion of an array of doubles, using a pre-computed mean.

The documentation states that `evaluate` takes five parameters: an array (`values`), an array (`weights`), the pre-computed mean (`mean`), a starting position (`begin`), and the number of elements to include (`length`). Furthermore, it stipulates that 0 is returned when `length=1`, and that `evaluation` calculates the variance with the following formula:

$$\text{variance} = \frac{\sum_{i=\text{begin}}^{\text{begin+length}} \text{weights}_i(\text{values}_i - \text{mean}^2)}{(\sum_{i=\text{begin}}^{\text{begin+length}} \text{weights}_i) - 1} \tag{5.7.1}$$

The implementation of this method contains a structural causal bug, as shown in Figure 5.8. During the computation of the denominator in Equation (5.7.1), the initialisation and condition of the `for` loop do not reference the specified start position, `begin`. Furthermore, the condition does not reference `length` to specify the correct end position, causing it to loop over the entire array instead. As a consequence, the causal effect of `begin` and `length` on the denominator is removed, and thus impacts the computed variance.

```
      double sumWts = 0;
-     for ( int i = 0 ; i < weights.length ; i++) {
+     for ( int i = begin ; i < begin + length ; i++) {
          sumWts += weights[i];
      }
```

Figure 5.8: A diff showing a structural causal bug located in the `evaluate` method of the `Variance` class.

Causal Software Graph

We constructed the CSG shown in Figure 5.9 using our limited domain expertise and the available documentation. From this we know that `mean`, `values`, and `length` are used to compute a "numerator", and that `length`, `begin` and `weights` should be used to compute a "denominator". These are then combined to yield the final `variance` result. There is a direct edge from `length` to `variance` to cover the specific case where length is either zero or where `begin+length` exceeds the length of `values`.

The `numerator` and `denominator` nodes are not immediately observable (hence the dashed boxes). When developing the model from the documentation, we start from an assumption that we are able to

[7]`https://github.com/AndrewC19/DAG-MT-case-studies`
[8]https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/stat/descriptive/moment/Variance.html

instrument the implementation to expose the corresponding values in the implementation as it executes (elaborated below).

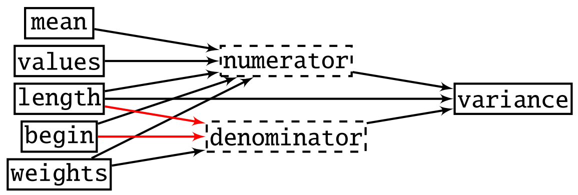


Figure 5.9: CSG for `Variance.evaluate`, with the fault-causing relationships highlighted in red and hidden variables shown as dashed boxes.

Metamorphic relations and tests

Using our CSG, we identified and manually wrote test cases for the 18 causal metamorphic relations (11 `ShouldCause` and 7 `ShouldNotCause`). Each test case specifies a control value and randomly samples a follow-up value using a series of basic generator methods.

In this instance, the process of specifying the CSG is likely to have required less effort than would be needed to specify these 18 metamorphic relations manually, particularly in the case of the `ShouldNotCause` relations. This is because the proposed approach automatically identifies which variables need to be controlled for in order to test each of the seven `ShouldNotCause` relationships - a process that would be time-consuming and error-prone if conducted manually.

Instrumentation

For the two hidden variables (`numerator` and `denominator`), we manually instrumented the code to enable us to get and set the value of these variables. To achieve this, we modified `evaluate` to include two new variables, `numerator` and `denominator`, that store the numerator and denominator shown in Equation (5.7.1), and updated the definition of `variance` to use these variables. We also return a hash map containing the variables in the CSG and their respective values. While manual instrumentation was trivial in this instance, we appreciate this would not be feasible for larger, unwieldy software systems, and that source code is not always available. We partially mitigate this limitation in the following case study.

Results

Out of the 18 metamorphic test cases, 16 passed and 2 failed on the buggy version of Apache Commons Math. The two failing test cases correspond to the fault-causing relationships highlighted in red in Figure 5.9: `length` → `denominator` and `begin` → `denominator`. We then ran the same test suite against the fixed version of Apache Commons Math, finding that all test cases passed as expected.

5.7.2 Apache Commons Lang

Next, we turn our attention to a bug in the `format` method of the `FastDateFormat` class, affecting version 2.5 of the Apache Commons Lang library (Bug ID: `Lang 26`). This is an efficient and threadsafe implementation for formatting and parsing dates that accounts for the user's locale[9].

A `FastDateFormat` instance can be created by calling one of its factory methods, `getInstance`, that takes three inputs: a string representation of a pattern (e.g. `"yyyy-mm-dd"`); a `TimeZone` instance that can be obtained given a string encoding a time zone ID (e.g. `"UK"`); and a `Locale` instance that is defined by two strings, one specifying the language (e.g. `"en"`) and the other specifying the country (e.g. `"US"`). Given a particular `date` as input, the `format` method is then called on the resulting `FastDateFormat` instance and returns a string representation of the specified date and format.

The implementation of the `format` method contains a structural causal bug, as shown in Figure 5.10. Here, `locale` is not passed to the constructor of the `GregorianCalendar` instance for which the date is formatted, resulting in the user's default system locale being used instead. Consequently, if `locale` is set to be different to the default system locale, `format` may return an unexpected string.

```
   public String format(Date date) {
-    Calendar c = new GregorianCalendar( timeZone );
+    Calendar c = new GregorianCalendar( timeZone, locale );
     c.setTime(date);
     return applyRules(c, new StringBuffer(maxLength)).toString();
   }
```

Figure 5.10: A diff showing a structural causal bug located in the `format` method of the `FastDateFormat` class.

This bug is problematic for locales that use the ISO 8601 date and time standard, such as `en-GB`, as this standard has 53 weeks for certain years (those with 53 Tuesdays), whereas others have 52. For example, if a `FastDateFormat` is instantiated with an ISO 8601 locale (e.g. `en-GB`) and the pattern `ww` (i.e week number) but the system locale is non-ISO 8601 (e.g. `en-US`), `format` will wrongly output week 1 instead of week 53 if the date is set to 1st January 2010.

Causal Software Graph

We constructed the CSG shown in Figure 5.11 using domain expertise and information available in the documentation. Nodes represent the variables used to instantiate the `FastDateFormat` object: `locale`, `timeZone`, and `pattern`. We then inserted two hidden variables (`rules` and `maxLength`) to capture the logic behind the initialisation of the `FastDateFormat` object. Specifically, the instance variables are used to construct a list of `rules` that are eventually used to create the formatted string (`dateStr`). The `rules` are also used to predict the maximum length of the formatted string (`maxLength`) to create a string buffer with sufficient capacity.

Metamorphic relations and tests

With the CSG in Figure 5.11 we constructed a total of 12 metamorphic relations, of which 7 are `ShouldCause` and 5 are `ShouldNotCause`. Using a series of basic generator functions, we automatically sample values for any variables that are a member of the (potentially empty) adjustment set of the metamorphic relation under test. A `FastDateFormat` object is then instantiated using each input configuration, before calling the `format` method with the fault-causing date, January 1st 2010.

[9] https://commons.apache.org/proper/commons-lang/javadocs/api-2.5/org/apache/commons/lang/time/FastDateFormat.html

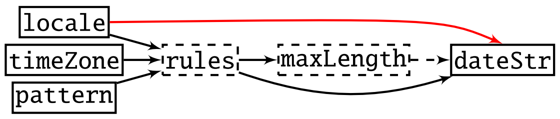


Figure 5.11: A CSG for the `FastDateFormat.format` method, with the fault-causing relationship highlighted in red, a misspecified edge shown by a dashed edge, and the hidden variables in dashed boxes.

Similar to the previous case study, by specifying a CSG we have likely reduced the amount of effort needed to identify these metamorphic relations manually, particularly in the case of the `ShouldNotCause` relations.

Instrumentation

Whereas the previous case study dealt with a single method, this case study concerns class-level behaviour. In particular, the hidden variables in our CSG (`rules` and `maxLength`) are derivable from class variables as opposed to internal method variables. This higher level of abstraction makes it feasible to observe and manipulate variables via bytecode manipulation, rather than manual instrumentation. The main advantage of using bytecode manipulation in this context is that it makes it possible to get and set the values of the key variables where access to the source code is limited.

To achieve this, using the javassist bytecode manipulation toolkit [156], we first added a statement to the end of the constructor which creates a new hash map instance variable to store key-value pairs for the variables and their values. Next, in order to enable us to intervene on the `rules` list, we added a statement that overrides the return value of `parsePattern`, a method which, according to the documentation[10], constructs the list of `rules` from the specified pattern. Finally, we add three statements to the `format` method: one at the start of the method which sets the `maxLength` value; another at the end of the method that stores the values of `pattern` and `maxLength` to the results hash map; and finally a call to a method that writes the results hash map to a properties file.

While this process of instrumentation is technically challenging and requires more up-front effort than manual instrumentation, it does not require the ability to manipulate the source code itself. Instead, tools such as javassist make it possible to write an instrumentation agent and class file transformer to conduct this kind of instrumentation with only a java class file and knowledge of the variable and method names of interest.

Results

Of the 12 metamorphic test cases, 10 passed and 2 failed. The failing test cases correspond to the fault-causing causal relationship highlighted in Figure 5.11, `locale` → `dateStr`, and a further unexpected failure in the relationship `maxLength` → `dateStr`. Contrary to our initial expectation, our tests reveal that `maxLength` has no causal effect on `dateStr`. After further investigation, we found that although `maxLength` determines the initial capacity of the `dateStr` string buffer, it has no observable impact on the contents of `dateStr` because the capacity of the string buffer is automatically expanded upon

[10]https://commons.apache.org/proper/commons-lang/javadocs/api-2.5/org/apache/commons/lang/time/FastDateFormat.html#method_summary

overflowing. However, we wrongly expected `maxLength` to set a hard upper-limit on the size of buffer that would result in an incomplete string.

Overall, in this case study, we were able to generate a set of metamorphic relations and tests that catch the structural causal bug in the `format` method of the `FastDateFormat` class. Additionally, these tests alerted us to a further discrepancy between our CSG and the implementation located in the `maxLength` → `dateStr` relationship that, upon investigation, turned out to be an error in our specification. Nevertheless, it was the process of generating metamorphic relations and tests from our CSG that revealed this misunderstanding.

5.8 Discussion

In this section, we discuss the main findings of this study, drawing on the results of both our experiment and illustrative case studies. We also discuss notable limitations of the approach proposed in this chapter.

5.8.1 Key Findings and Contributions

The main contribution of this chapter is the introduction of a causal model-based approach to the generation of metamorphic relations. By drawing on the inherently causal nature of metamorphic relations that we outlined in earlier chapters (see Chapter 3), we have demonstrated how metamorphic relations can be systematically constructed from lightweight and domain agnostic models of causality. This provides further evidence that metamorphic testing is indeed a causal testing activity that can benefit from being framed and solved as a problem of causal inference.

The second contribution is a controlled experiment in which we evaluated the performance of our technique with respect to two aspects: (1) robustness to misspecification, and (2) robustness to 'evasive' causal relationships. The results of our experiment demonstrate that, even with moderate amounts of misspecification (i.e. addition of non-existent edges or deletion of existing ones), the approach is still able to detect bugs that affect the causal structure of the PUT. This indicates that the approach does not require a perfect causal specification to be of benefit to a tester. Moreover, our results show that, as expected, when the PUT's constituent causal relationships are made to be increasingly evasive (i.e. hard to detect), the performance of the approach deteriorates. However, we also show how using larger randomly generated test suites can effectively mitigate this challenge. This suggests that more advanced test generation strategies could be paired with our technique to increase its performance.

The third contribution is a pair of illustrative case studies in which we applied the approach to detect two structural causal bugs in Java programs from the Defects4J framework [17]. Until this point, our approach had only been applied to the synthetic programs used for the experimental evaluation. To this end, our case studies show how the proposed approach can be applied to actual programs implemented in a different programming language and give concrete examples of actual bugs that affect causal structure. In particular, this involved instrumenting each PUT to expose hidden variables that were not accessible by default. This required additional effort that would, in practice, pose a barrier to applying our technique. We discuss the limitations surrounding this work further in the following sub-section.

5.8.2 Limitations

While our experimental evaluation and illustrative case studies provide evidence for the performance and applicability of the proposed approach, there are several limitations that are worth noting.

Specifying a CSG

While our solution essentially reduces the problem of specifying metamorphic relations to drawing a causal DAG, this process incurs its own overhead; for complex and unwieldy systems, drawing a causal DAG may be a non-trivial and time-consuming task. The exact cost of constructing a causal DAG is hard to quantify. However, the increasing adoption of causal DAGs as a lightweight domain model [152] across a range of fields such as epidemiology [157, 158] indicates that the effort involved is perceived to be low enough to justify their use. Furthermore, causal DAGs are supported by an extensive mathematical framework that alleviates the burden associated with identifying independence relations - a task that, if conducted manually, quickly becomes cumbersome as the size of the subject system increases.

Ideally, the cost of our approach could be reduced by utilising automatic approaches for generating causal structures. For example, the field of causal discovery [159] offers the potential to learn causal models from existing data. Alternatively, we could turn to program dependence graphs as used by the causal inference-led approaches to fault localisation discussed in Section 2.3.1. However, both approaches share the same fundamental limitation: they rely on using artefacts derived from the current implementation of the PUT as a model of causality. This means that the resulting causal structure is a representation of the actual behaviour of the PUT, not the expected behaviour. Therefore, we would, in effect, be trivially testing the PUT against itself. Nonetheless, future work could investigate whether such automated approaches could be helpful in providing a starting point for a causal structure.

Instrumentation

A key barrier to applying our technique is that it requires the ability to intervene on any node in the CSG. Since most programs have internal states that may be of interest for testing but are not accessible by default, in practice, it is generally necessary to apply some form of instrumentation or testability transformation. This increases the overall cost associated with using the approach and may make it harder to integrate with existing testing infrastructure.

To address this issue, future work could investigate using more advanced causal inference methods that are capable of handling missing variables or data. For example, a form of causal inference known as instrumental variables can be used to overcome the problem of unobserved confounding [145]. This offers the potential to bypass hidden variables provided additional assumptions can be met.

Misleading Causal Tests

In the presence of evasive causal relationships (i.e. those that are hard to detect), we can encounter instances where our metamorphic relations pass or fail misleadingly. Suppose we have a relationship $X \to Y$ in the PUT. A metamorphic test for independence ($X \perp\!\!\!\perp Y$) may wrongly pass if we do not happen to intervene on X in a way that changes the value of Y, and a metamorphic test for $X \to Y$ may wrongly fail for the same reason. More generally, where a program is non-injective, we can only conclusively prove $X \perp\!\!\!\perp Y$ if we exhaustively test the input space of X (i.e. performing all possible interventions).

Since exhaustive testing of the input space is generally infeasible, to mitigate this limitation, we should instead aim to construct test suites that are effective at detecting the evasive causal relationships at the centre of this problem. With this in mind, future work could evaluate whether existing, more advanced test generation strategies could be employed to generate metamorphic tests that are better able to detect evasive causal relationships. Moreover, future work could also investigate whether novel test adequacy criteria would be beneficial for guiding such test generation strategies to detect evasive causal relationships.

5.9 Concluding Remarks

While metamorphic testing is a well-established solution to the test oracle problem [15, 13], a key challenge is to devise metamorphic relations capturing the expected behaviour of software in terms of how particular *changes* to the input parameters should bring about changes to the outputs. As outlined in Chapter 2, this is typically a manual and time-consuming process that relies on significant domain expertise [43]. Furthermore, in general, existing techniques for the identification of metamorphic relations rely heavily on the user to identify domain-specific relations [45], or infer relations using the program itself [51], meaning the inferred relations may reflect bugs in the implementation of the PUT.

Motivated by these problems and the wider objective of understanding how causal inference can be of benefit within a software testing context, in this chapter, we proposed a model-based technique for identifying metamorphic relations from a lightweight, intuitive graphical causal model of software behaviour. This techniques effectively shifts the problem of identifying metamorphic relations to one of specifying a causal DAG that captures the expected causal relationships between variables of the PUT. We evaluated our technique using synthetic programs with a known causal structure, finding that our technique is able to produce useful metamorphic relations and tests capable of catching bugs that affect causal structure (coined as structural causal bugs), even in the presence of misspecification and evasive (i.e. difficult to observe) causal relationships.

To demonstrate the applicability of our technique to real systems, we also applied it to a pair of Java programs from Defects4J with known structural causal bugs. This indicated that our technique is not only able to catch structural causal bugs at various levels of abstraction, but it can also alert the user to problems with the specification. Moreover, these examples show how instrumentation can be employed to 'open the black-box' and test causal relationships involving variables that are not immediately accessible via the interface of the PUT (see Definition 4.1).

In future work, we plan to enrich the proposed graphical model with functional information, such that we can also generate metamorphic relations capable of catching bugs that alter the functional form of a causal relationship without manipulating causal structure. There is also the potential to combine the proposed technique with more advanced test generation and causal inference techniques to improve the ability to detect evasive causal relationships and bypass hidden variables, respectively. Moreover, in order to draw more generalisable conclusions about performance, a more comprehensive evaluation and application to further case studies is desirable.

Briefly returning to our problem statement outlined in Section 3.4, the experimental evaluation and illustrative case studies have shown how causal inference techniques can be leveraged to lessen the burden associated with the identification of metamorphic relations and, in turn, lower the barrier to entry for metamorphic testing. Therefore, this chapter has provided evidence to support our argument that metamorphic testing is a fundamentally causal task that can benefit from being viewed and solved as a problem of causal inference.

To summarise, in this chapter, we have:

- Developed a systematic and domain agnostic model-based approach for the identification of metamorphic relations using graphical causal inference techniques.
- Evaluated the ability of the approach to detect bugs affecting causal structure (i.e. adding spurious causal relationships or removing genuine ones) using synthetic programs generated from a ground-truth causal structure.
- Demonstrated the wider applicability of the technique by applying it to a pair of actual Java programs from the Defects4J framework which contain known structural causal bugs.

In the following chapter, we turn our attention to the second problem outlined in our problem statement from Section 3.4: the cost associated with executing metamorphic tests, particularly in the context of costly non-deterministic systems such as computational modelling software. Drawing upon the conceptual CTF from Chapter 4, we again focus on exploring how causal inference methods can help to address this challenge and, in particular, establish how observational methods of causal inference can be of benefit in this setting.

6 | Metamorphic Testing with Observational Data

In this chapter, we turn our attention to the remaining part of our problem statement that we have yet to address: the cost of executing metamorphic test cases. As a brief reminder, the complex and often non-deterministic nature of computational modelling software like Covasim (our second motivating example introduced in Section 3.2), makes metamorphic testing a costly and potentially impractical task. Yet, due to the important applications that are often associated with this class of software, testing is an extremely important task that should not be overlooked.

To address this challenge, we build upon the conceptual CTF introduced in Chapter 4 to investigate whether graphical causal inference methods can be used to infer metamorphic test outcomes from existing data. We approach this task from a model-based perspective, in which we use the components of our causal specifications - namely, a causal DAG and modelling scenario - as a basis for designing test cases that can derive metamorphic test outcomes from both interventional and observational data. This offers an alternative approach to testing causal relationships in prohibitively expensive forms of software such as computational models, where conventional metamorphic testing is impractical. To achieve this, we exploit the fundamentally causal nature of metamorphic testing that we discussed informally in Section 4.1 and later defined within our conceptual CTF in Section 4.2.

We start by providing a Python reference implementation of the CTF previously introduced in Chapter 4. We then apply our implementation of the CTF to a series of real world computational models from different domains, including the epidemiological agent-based model that formed our second motivating example in Chapter 3, to evaluate the accuracy, efficiency, and practical effort involved in using the approach.

This chapter is based on the case studies from publication **P3**: *Testing Causality in Scientific Modelling Software* (TOSEM, 2023).

6.1 Introduction

As demonstrated in Chapter 3, metamorphic testing provides a solution to the test oracle problem, making it well-suited for testing computational models *in theory*. However, in practice, computational models have a number of challenging characteristics that make them particularly costly subjects to execute and thus test [13]. In this chapter, we are specifically concerned with this intrinsic challenge: How can we apply metamorphic testing to a system with a vast and complex input space, which may be non-deterministic and suffer from the test oracle problem, without the ability to resort to large numbers of test executions?

To address this challenge, we return to the motivation outlined in Chapter 4 to explore whether well-established observational causal inference techniques can provide a cost-effective alternative to conventional metamorphic testing. The concept behind the approach presented here is to create a causal model capturing some behaviour of the computational modelling software that is of interest for testing. By abstracting away unnecessary details, this causal 'model-of-the-model' then provides a more cost effective means to test a wide array of metamorphic relations using observational causal inference methods. If the model-of-the-model is accurate, these metamorphic relations should *also* hold for the original computational modelling software, with reasonable accuracy.

To explore and evaluate this approach, we provide a reference implementation of the conceptual CTF introduced in Chapter 4 and apply it to a pair of real-world computational models. While we focus on evaluating the benefits of the observational approach to causal inference in this setting, we also demonstrate how the framework enables metamorphic testing to be achieved experimentally (i.e. in the conventional way by executing the computational model - rather than a causal model-of-the-model - under the conditions necessary to isolate the causal effect of interest).

Overall, we make three contributions in this chapter:

1. We introduce a reference implementation for the conceptual CTF introduced in Chapter 4 that facilitates the application of observational causal inference methods to metamorphic testing.
2. We apply the proposed framework to a series of case studies involving real-world computational models from different domains, evaluating its ability to predict metamorphic test outcomes from observational data in terms of accuracy, cost, and effort.
3. We present empirical evidence demonstrating that metamorphic test outcomes can be predicted from significantly less data than is required by conventional metamorphic testing approaches.

In addition, by applying metamorphic testing through our reference implementation of the CTF, we identify a notable bug related to vaccination in Covasim (our motivating ABM example from Section 3.2) that was later acknowledged and fixed by the developers.

The remainder of this chapter is structured as follows. In Section 6.2, we outline the steps involved in an observational causal inference-led approach to metamorphic testing. Then, in Section 6.3, we outline our reference implementation of the CTF, before applying it to real-world computational models to demonstrate how it can be used to perform metamorphic testing using both interventional and observational data in Section 6.4. Following this, we discuss the main findings and threats to validity in Section 6.5. Finally, we conclude the chapter in Section 6.6.

6.2 Metamorphic Testing using Observational Causal Inference

In Section 4.3, we showed how metamorphic testing could be expressed as a problem of causal inference within the CTF using interventional data. That is, data generated for the express purpose of the metamorphic test in question. In this section, we turn our attention to one of the main prospective benefits of a causal inference-led approach to metamorphic testing: the ability to predict metamorphic test outcomes from existing test data using observational causal inference. This section outlines how, using observational data, the components of the CTF can be used to perform the main steps of causal inference outlined in Section 2.2.1: framing the causal question, identification (Section 2.2.6), and estimation (Section 2.2.7).

6.2.1 Framing the Causal Question

The first step of causal inference is to frame the causal question of interest. In line with the consistency assumption (see Section 2.2.3), this requires a clear specification of the intervention of interest and the population to which it should be applied. From a metamorphic testing perspective, this causal question is implicitly encoded by the metamorphic relation of interest. For example, as previously mentioned in Section 4.1, the metamorphic relation $sin(x) = sin(\pi - x)$ is essentially asking the question: *Does subtracting the input x from π cause any change to the output of sin?* Thus, this step is already handled by specifying the metamorphic relation (Definition 2.1) of interest.

Within the CTF, these fundamental components are captured by a causal test case (Definition 4.5). First, to capture the intervention of interest, the causal test case contains an intervention Δ. Second, to capture the population of interest, the causal test case contains an input valuation $\bar{\mathbf{I}}$ that defines the precise input setting that the intervention should be applied to. In addition, a modelling scenario (Definition 4.4) is specified that places constraints over input variables in order to characterise the precise type of scenario to which the causal test applies, further characterising the population of interest.

Consequently, a causal test case can be executed directly in order to obtain interventional data (by performing the specified intervention on the specified population). Alternatively, where the objective is to infer causal effects from observational data, the modelling scenario can be used to filter any irrelevant data. That is, data that does not fall within the specified modelling scenario.

6.2.2 Identification

At this point, we have specified the target causal question and obtained relevant data. Where the data is interventional, this step can be skipped, as it has been collected under the conditions necessary to isolate the causal relationship of interest. However, where the data is observational, we must identify potential sources of bias in order to satisfy the (conditional) exchangeability condition (see Section 2.2.3). This step is referred to as identification.

As explained in Section 2.2.6, identification involves identifying a set of variables that, once adjusted for, mitigate all forms of bias. This can be performed by applying the back-door criterion (Definition 2.5) to the causal DAG and is the key step in ensuring that the exchangeability condition is satisfied. Ultimately, it is this step that enables us to express our target causal quantity (e.g. Equation (2.2.2)) as a closed-form statistical expression (e.g. Equation (2.2.6)).

Consequently, we require users to provide a causal DAG representing the key cause-effect variables in the modelling scenario of interest as part of the causal specification (Definition 4.6). By performing identification on this causal DAG, we arrive at an adjustment set that helps to inform the design of our estimator in the following step. In addition to facilitating the process of identification, this graphical artefact also makes the causal assumptions responsible for exchangeability transparent.

6.2.3 Estimation

Following identification, we possess a ‘recipe’ for a statistical procedure that can infer causal estimates from our observational data. All that remains is to implement this procedure. This corresponds to ‘adjusting for’ the identified biasing variables and requires the specification of a statistical model capable of performing the adjustment, such as a regression model.

In this chapter, we focus on regression models over other statistical methods, such as standardisation, as it enables the user to address positivity violations in data (see Section 2.2.3). More specifically, where the data is incomplete, a well-specified statistical model (i.e. one that accurately depicts the underlying functional form of its subject) can borrow information from similar data in order to smooth over gaps and thus mitigate positivity violations. Therefore, where the data is incomplete, by leaning on the assumptions encoded by a regression model, we can (approximately) satisfy the remaining assumption that is necessary for valid causal inferences. The resulting regression model can then be used to predict how the model would respond to unseen interventions (i.e. metamorphic test outcomes that have not been observed).

Now that we have outlined how metamorphic test outcomes can be inferred from observational data, in the following section, we outline our reference implementation of the CTF that will be used to implement this process.

6.3 Implementation

In this section, we outline our open-source Python reference implementation of the CTF,[1] comprising over 4000 lines of Python code. This comprises four key stages that are summarised in Figure 6.1: Specification, Test Cases, Data Collection, and Testing.

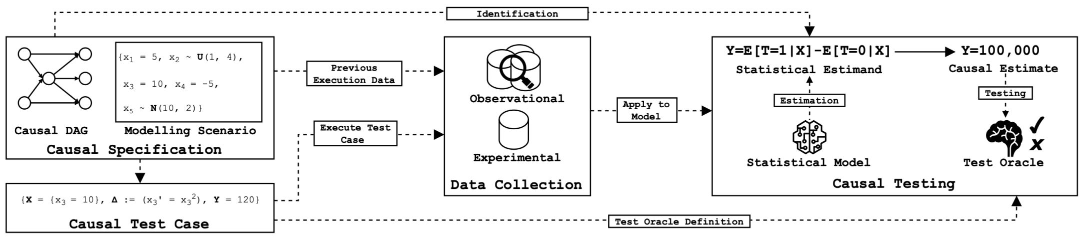


Figure 6.1: Workflow of the CTF reference implementation.

6.3.1 Causal Specification

To begin causal testing, we form a causal specification (Definition 4.6), comprising two components: a modelling scenario (Definition 4.4) and a causal DAG (Definition 2.2). We form the modelling scenario by specifying a set of constraints over the inputs that characterise the scenario-under-test, such as $I_1 < I_2$. In line with Definitions 4.4 and 4.8, these constraints are used as the basis for determining whether data is *valid* or *invalid*. Next, we specify our causal DAG using the DOT language [160], in which graphs are expressed as a series of edges, such as $I_1 \rightarrow I_2$.

[1]https://github.com/CITCOM-project/CausalTestingFramework

Although there are no set guidelines for the process of constructing a causal DAG, we generally take the following steps. First, we start by constructing a complete directed graph over the interface of the PUT (i.e. $\mathbf{I} \cup \mathbf{O}$ from Definition 4.1). Then, to simplify this structure, we apply the following assumption:

Assumption 1. Outputs cannot cause inputs.

Assumption 1 follows from temporal precedence (that a cause must precede its effect) [161] and the observation that, in a given test execution, outputs temporally succeed inputs. This enables us to delete all edges from outputs to inputs.

Then, in many cases, we can also apply the following assumption to remove all edges from inputs to inputs:

Assumption 2. Inputs cannot cause changes to the values of other inputs and, therefore, cannot share causal relationships.

Assumption 2 follows the observation that, in general, all inputs are assigned their values prior to execution. Under this characterisation, changes to the value of one input cannot *physically* affect another input's value and, therefore, inputs cannot share causal relationships. Of course, there are caveats to this; if a system has input validation, for example, the assignment of one input's value may *physically* restrict which values can be selected for a second input. Note that, in such cases, our framework is still applicable, but the user would have to consider more edges manually to construct their DAG.

6.3.2 Causal Test Case

Now that we have a causal specification, we define a causal test case (Definition 4.5) that describes the intervention (Definition 4.3) whose effect we wish to test in the context of a particular modelling scenario. In our reference implementation, a causal test case is an object that requires us to specify a control input configuration, a treatment input configuration, and an expected effect. Here, the treatment input configuration is the outcome of applying the intervention to the control input configuration. In the following steps, this information will enable us to collect appropriate test data (Data Collection), design quasi-experiments isolating the causal effect of interest within this data, and define test oracles that ascertain whether the expected causal effect is observed (Causal Testing).

6.3.3 Data Collection

After creating a causal specification and causal test case, the next step is to collect data corresponding to the modelling scenario. We can achieve this using two distinct approaches that roughly align with experimental and observational approaches to causal inference. First, in situations where it is practical or desirable to directly execute the PUT, we can take a (quasi-)experimental approach, executing the PUT under the conditions necessary to isolate the causal effect of interest. Second, in situations were it is impractical to execute the PUT but have access to prior execution data, we take an observational approach in which we infer the causal effects of interest from existing data. To facilitate these two approaches, our reference implementation provides two modes of data collection.

Experimental Data Collection

Experimental data collection executes the model *directly* under both the control and treatment input configuration to isolate the causal effect of the intervention. To this end, our reference implementation provides an abstract experimental data collector class, requiring the user to implement one method that executes the model with a given input configuration. In essence, this requires the user to implement a test harness that can take a dictionary of input settings, run the model with those settings, and return the resulting outputs as a CSV. Using this test harness, the CTF then executes the model under the control and treatment input configurations specified by the causal test case to collect *interventional data* (see Section 6.2) from which we can measure the causal effect directly (i.e. without adjusting for various forms of bias).

Observational Data Collection

Since it is often infeasible to run computational models many times, we also provide the option to infer test outcomes from observational, existing test data. This data may not meet the experimental conditions necessary to isolate the causal effect and thus may contain biases that lead purely statistical techniques astray. However, by employing graphical causal inference techniques, the CTF can identify and mitigate bias in the data, providing an efficient method for testing computational models *a posteriori*.

There are two caveats to this. First, the causal DAG must be correctly specified. While this is not generally verifiable, several techniques exist that can quantify the sensitivity of causal estimates to unobserved confounding, including the robustness value [162] and the e-value [163]. These techniques could be employed to justify that the causal DAG-informed adjustment set yields causal estimates that are robust to missing confounders. Second, the observational data must be consistent with the constraints of the causal specification. To this end, our reference implementation includes an observational data collector class that takes a CSV file of existing test data as input and uses the Z3 theorem prover [164] to identify and remove any model runs that violate constraints. Next, we describe how the CTF infers test outcomes from this data.

6.3.4 Causal Testing

Given a causal test case and causal specification, testing is carried out in two stages: causal inference and applying the test oracle.

Causal Inference

To infer the causal effect of interest, our reference implementation applies the two steps of causal inference outlined in Section 2.2: identification and estimation.

Identification As discussed in Section 2.2.6, identification refers to the process of identifying sources of bias that must be adjusted for in order to satisfy (conditional) exchangeability (see Equation (2.2.4)). This is an essential step in causal inference as it is the exchangeability assumption that enables us to take a purely associational measure (e.g. the ATE; see Equation (2.2.6)) and treat it with causal status. In other words, under exchangeability we are able to reverse the adage ‘association is not causation’ to ‘association *is* causation’.

Causal DAGs are particularly helpful in performing identification as they are supported by an extensive range of mathematical tools and techniques that can, amongst other things, automatically derive or validate adjustment sets. Leveraging these abilities, the CTF offers two options for identification. First, the user can manually specify an adjustment set for the effect of interest. A method is provided that can be used to check whether the selected adjustment set satisfies the back-door criterion (Definition 2.5) - that is, whether it is a valid adjustment set. In addition, the CTF provides a method for automatically identifying the *minimal* adjustment set for a particular edge in the causal DAG. That is, the smallest set of variables that satisfies the back-door criterion (of which there may be none or many). This implements a polynomial time algorithm for listing minimal adjustment sets proposed by Textor and Lískiewicz [165].

Estimation Following identification, we take the adjustment set and design an appropriate statistical estimator that adjusts for these variables, and apply this estimator to our data to estimate the desired causal metric (e.g. ATE or RR). Our reference implementation provides default regression and causal forest [166] estimators which can be customised to add additional features such as squared and inverse terms to change the shape of the model. This enables users to configure models based on their domain expertise in order to achieve more accurate inferences and overcome positivity violations by interpolating from the shape of the regression surface to smooth over gaps in the data.

In addition, the CTF includes an abstract estimator class that enables users to define their own estimators. This requires the user to implement two abstract methods. The first method involves updating a list of modelling assumptions that keep track of the assumptions carried by the particular statistical model. For example, if a regression model is implemented that includes an interaction term (see Section 2.2.8), it carries the assumption that the effect interacts with another variable. The second method requires the user to estimate the effect measure of interest, such as ATE or RR, using the selected model. Together, these two methods allow users to expand on the basic estimators in a flexible way, while ensuring the modelling assumptions that are essential for causal inference are also made transparent.

This step outputs a causal test result containing the inferred causal estimate for the desired causal metric (e.g. ATE or RR) and 95% confidence intervals. The user is, of course, free to relax their confidence intervals should they wish to obtain a more precise estimate with a higher level of variance, or vice versa.

Test Oracle

After applying causal inference and obtaining the causal test result, all that remains is the causal test oracle (Definition 4.7). That is, the procedure used to check whether the causal test results match the expected outcome of the causal test case. For this purpose, our reference implementation provides several pre-defined test oracles that check for positive, negative, zero, and exact effects. For example, the positive and negative test oracles check whether the causal estimate is postive/negative and whether the confidence intervals contain zero. If the confidence intervals contain zero, this means there is not a statistically significant positive/negative effect at the selected confidence level, causing the test to fail.

Alternatively, to handle more complex outputs, a user can specify a custom test oracle that ascertains whether a causal test result should pass or fail. This requires the user to implement a single abstract method that takes the outputs of a causal test case (namely, the causal effect estimate and confidence intervals), and return a Boolean value denoting whether the test has passed or failed, based on these outcomes.

Now that we have discussed the workflow of our CTF reference implementation, in the following section, we demonstrate its application to a pair of real-world computational models in three different metamorphic testing scenarios.

6.4 Case Studies

In this section, we apply the CTF to two real-world scientific models from different domains, approaching metamorphic testing as a CI problem. Our goal here is to conduct a series of *evaluative* case studies [167] that appraise the CTF with respect to three attributes: *accuracy*, *efficiency*, and *practicality*. Here, we do not aim to draw generalisable conclusions, but to evaluate the CTF with respect to each of these attributes within the context of each subject system. To achieve this, across our case studies, we corroborate evidence to collectively answer the following research questions:

RQ1 (Accuracy): Can we reproduce the results of a conventional metamorphic testing approach by applying the CTF to observational data?

Causal inference is a generally applicable technique [168] that promises the ability to infer test outcomes from existing data that is potentially confounded. In the context of testing computational modelling software, this approach has the potential to reduce the overhead associated with metamorphic testing and SMT by enabling the inference of metamorphic test outcomes from existing execution data. This is in contrast to a conventional approach which may require numerous potentially costly executions.

In this research question, we consider whether the CTF is able to predict metamorphic test outcomes from observational data with sufficient accuracy to make *actionable inferences*. By actionable inferences, we refer to predicted outcomes that provide a truthful and meaningful insight into the actual behaviour of the SUT.

RQ2 (Efficiency): In terms of the amount of data required, is the CTF more cost effective than a conventional metamorphic testing approach?

In practice, the utility and applicability of the CTF depends on the amount of observational data required to make actionable inferences. Hence, for the CTF to be considered a useful tool and a viable alternative to conventional metamorphic testing and SMT approaches, it should be capable of making actionable inferences using no more data than is required by a conventional approach.

To this end, in order to understand the efficiency and therefore utility of the proposed approach, this research question investigates the relationship between the amount of observational data and the accuracy of insights provided by the inferred metamorphic test outcomes.

RQ3 (Practicality): What practical effort is required from the tester to conduct metamorphic testing using the CTF?

The CTF requires causal knowledge and domain expertise that, in turn, depend on human effort. This human effort cannot be overlooked. Hence, in order to determine whether the technique can be considered practical and applicable, it is necessary to investigate the trade-off between the human cost and the benefits offered by the CTF.

In this research question, we provide a qualitative account of the human effort involved in applying the CTF to each case study.

In the remainder of this section, we cover each case study in accordance to the following high-level structure. First, we describe the characteristics of the subject system and our justification for selecting it. We then provide a brief overview of the testing activity (the broad testing objective) and the process of acquiring data for analysis. Following this, we describe the application of the CTF and analyse the generated data. We conclude by analysing the outcomes and answering the relevant research questions. The contribution of each case study to the research questions will be highlighted throughout the case studies and the collective findings will be discussed in Section 6.5.

6.4.1 Cardiac Action Potential Model

In this case study, we use the CTF to conduct sensitivity analysis on the Luo-Rudy 1991 ventricular cardiac action potential model [169] (LR91) in a straightforward and efficient way. Sensitivity analysis is commonly used to validate and verify scientific models, with a specific focus on identifying which inputs have the greatest impact on model outputs [170, 171]. Here, we take a causal inference-led approach and measure the ATE of several input parameters on one output, quantifying the extent to which this output is affected by changes to the inputs. As test oracles, we construct a series of metamorphic relations that capture the expected magnitude and direction of each ATE.

Throughout this case study, we follow part of an existing study [172] that conducts uncertainty and sensitivity analysis on LR91 using a **Gaussian Process Emulator** (GPE) [173] trained on runs of the model. This work provides an invaluable source of domain expertise that precisely quantifies several relationships between the inputs and outputs of LR91 that we use as the basis for constructing our metamorphic relations. However, in contrast to the data-driven approach employed in the original study, we employ causal knowledge and domain expertise to justify and hand-craft a simple regression model that reaches the same conclusions. The code for reproducing this case study can be found in our open source repository[2].

Subject System

The LR91 ventricular cardiac action potential model [169] is a mathematical model comprising a system of differential equations that describe the rapid rise and fall in the voltage across the membrane of a mammalian ventricular cell. This characteristic rise and fall in voltage is referred to as an *action potential*. The behaviour of this model is controlled by 24 constants, 8 rate variables, 8 state variables, and 25 algebraic variables.

We selected LR91 as a case study as it follows a different modelling paradigm to our other subject system and has supported extensive and important research into cardiovascular physiology. Furthermore, amongst its vast and largely uncertain input space, LR91 has several well-characterised input-output relationships suitable for causal analysis.

An example action potential produced by LR91 is shown in Figure 6.2, demonstrating the rapid rise (known as depolarisation) and corresponding fall (repolarisation) of the voltage over time. In this case study, we quantify the effect of six conductance-related input parameters on one attribute of the action potential: *action potential duration to 90% of repolarisation* (APD_{90}). That is, the amount of time taken for the action potential to repolarise by 90%. This output concerns the falling phase of the action potential in which the cell returns to its resting voltage [174] and is shown in Figure 6.2.

[2] https://github.com/CITCOM-project/CausalTestingFramework/tree/683e6c55/examples/lr91

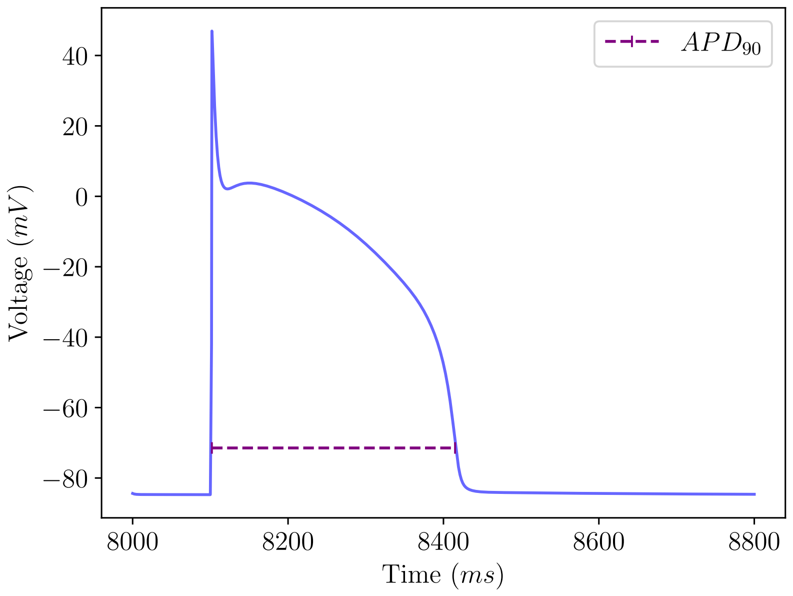


Figure 6.2: An example action potential produced by the Luo-Rudy 1991 model, simulating the rise and fall of voltage across a mammalian ventricular cell, and the output of interest: APD_{90}.

Testing Activity

In line with the approach followed by Chang et al. [172], we trained a GPE on 200 runs of LR91, with input configurations sampled via Latin Hyper Cube Sampling [175] from a series of normalised uniform design distributions to ensure even coverage of the input space. The GPE was then used to calculate the expectation of a given output, conditional on an input of interest, to quantify the effect of varying each of the six inputs on the eight output parameters, over the range of the design distribution.

From a causal inference perspective, we can obtain similar information by computing the ATE of each input on each output over the range of the design distribution. Specifically, we can set our control value to the mean value of the design distribution and uniformly increment our treatment value from the minimum to the maximum value of the design distribution. This yields a series of ATEs that quantify the expected change in output caused by changing the input parameters by specific amounts above and below their mean, revealing the magnitude of each input's effect on the outputs.

Here, we focus our analysis on the effects of the six inputs on one output, APD_{90}. We have selected this output because the original paper uses it to illustrate the approach. Based on the results reported in [172], we expect the following metamorphic properties to hold:

1. Increasing the parameters G_K, G_b, and G_{K1} should cause APD_{90} to decrease.
2. Increasing the parameter G_{si} should cause APD_{90} to increase.
3. Increasing the parameters G_{Na} and G_{Kp} should have no significant effect on APD_{90}.

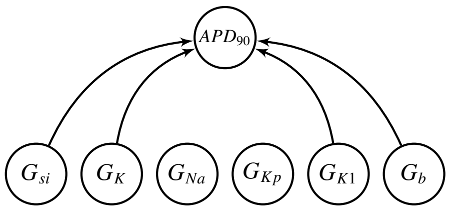


Figure 6.3: LR91 modelling scenario's Causal DAG, where the sensitivity of APD_{90} to each conductance input is computed as the causal effect (ATE).

4. The following monotonic relationship should hold over the (absolute) magnitude of the parameters' effects:

$$|APD_{90}^{G_{si}}| > |APD_{90}^{G_K}| > |APD_{90}^{G_b}| > |APD_{90}^{G_{K1}}|$$

Data Generation

To gather data from LR91, we followed the same approach as Chang et al. [172], where the 200 input configurations were sampled from the design distributions using Latin Hyper Cube sampling and then normalised. We then executed each of these input configurations on an auto-generated Python implementation of LR91 from the cellML modelling library [176]. We extended this implementation to enable us to sample the input values via Latin Hyper Cube sampling and automatically extract the outputs.[3]

Causal Testing

To approach metamorphic testing as a causal inference problem, we first specify our modelling scenario and causal DAG. For this set of tests, the modelling scenario constrains each input to the range of its uniform design distribution (as specified in the original paper [172]):

$$\{17.250 \leq G_{Na} \leq 28.750, 0.0675 \leq G_{si} \leq 0.1125, 0.2115 \leq G_K \leq 0.3525,$$
$$0.4535 \leq G_{K1} \leq 0.7559, 0.0137 \leq G_{Kp} \leq 0.0229, 0.0294 \leq G_b \leq 0.0490\}$$

As in the original study, these input values were then normalised to the range [0, 1].

We then specify the expected cause-effect relationships (and absence thereof) as the causal DAG shown in Figure 6.3. While not essential, we include the isolated nodes G_{Na} and G_{Kp} in our DAG to make our expectation for the absence of a causal effect explicitly clear. For each relationship, we then create a suite of causal test cases covering a series of interventions that incrementally increase/decrease the value of the inputs over the range of the design distribution. For each input, this is achieved by setting the control value to 0.5 (the mean) and uniformly sampling 10 treatment values over the range [0, 1]. This produces a total of 10 test cases per input that vary its value from 0.5 to each of the treatment values: [0, 0.1, 0.2, ... 1.0]. Using the CTF, we then perform identification and estimation. Here, the cause-effect relationships are straightforward and there is no confounding to adjust for, enabling us to fit a regression model $APD_{90} \sim x_0 + x_1 G_z$ for each input $z \in \{si, K, Na, Kp, K1, b\}$. Using these models, we then predict the ATE and 95% confidence intervals for each test.

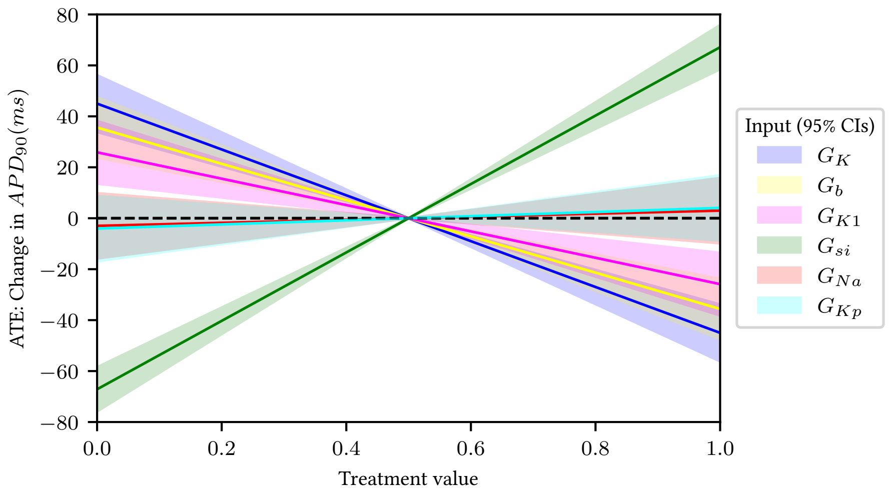


Figure 6.4: Sensitivity of APD_{90} in response to changes to the mean value of input parameters in LR91.

Results

The results, as summarised in Figure 6.4, show that all expected metamorphic relationships pass with statistical significance (95% confidence intervals do not contain 0) and are visually similar to Figure 5 in the original study [172]. Specifically, the first metamorphic relation holds as G_K, G_{K1}, G_b have negative effects, the second metamorphic relationship holds because G_{si} has a positive effect, and the third metamorphic relation holds as G_{Na} and G_{Kp} have no significant effect. Furthermore, the fourth metamorphic relation holds as the gradients corresponding to these effects reveal that the effect sizes follow the expected monotonic relationship: $|APD_{90}^{G_{si}}| > |APD_{90}^{G_K}| > |APD_{90}^{G_b}| > |APD_{90}^{G_{K1}}|$.

This case study has provided insights into **RQ1** and **RQ3**. As a result of following the data generation approach of the original paper, however, this case study did not afford us the opportunity to evaluate the efficiency of the CTF.

RQ1 Accuracy In this case study, we used the CTF to conduct a sensitivity analysis on the LR91 model, achieving visually similar results to an existing approach that employed a GPE [172]. However, we achieved this using a significantly simpler statistical model whose design was informed by causal reasoning as opposed to associations within the data. This contrast between a model-based and black-box approach to reasoning about system behaviour raises an interesting discussion around *explainability* that we return to in Section 6.5.

RQ3 Practicality In this case study, the process of specifying the causal DAG was straightforward and required minimal domain expertise that were easily gleaned from the original study [172]. Since the resulting DAG contained no confounding (Figure 6.3), the regression model for each causal test simply regressed the input-under-test against against APD_{90}. By contrast, Gaussian Processes (as used in the original study) have several practical limitations, including the need to specify an appropriate kernel

[3]Our LR91 model is available at: https://github.com/AndrewC19/LR91/tree/769e7ff

function for the problem at hand [177], and a complexity of $O(n^3)$ that hinders the feasibility of the approach when dealing with large amounts of data [178].

Overall, in this case study, we have shown that the CTF reaches the same conclusions as the original study. However, the CTF achieves this by using a simpler, more practical statistical model guided by causality instead of associations within the data.

6.4.2 Covasim: Experimental Causal Testing

In this case study, we demonstrate the ability of the CTF to conduct SMT of Covasim [127] using the experimental mode of the CTF (Section 6.3.3). That is, isolating the causal effect of interest via strategic executions of the PUT, rather than applying graphical causal inference to observational data. Our aim here is to provide evidence to support our claim that metamorphic testing is a fundamentally causal activity that can be framed and solved as a problem of causal inference. The code for this case study can be found in our open source repository[4].

Subject System

As a reminder, Covasim is the epidemiological ABM that was introduced as a motivating example in Section 3.2. We cover two testing scenarios using Covasim. In this section, we elaborate upon our running example from Section 4.2.2 (Examples 4.3 to 4.6, 4.9 and 4.10) and use the experimental mode of the CTF to test the effect of prioritising the vaccination of elderly people on several vaccine-related outcomes, revealing an interesting bug in the process. Then, in Section 6.4.3 we test the effect of increasing the β parameter (transmissibility) on cumulative infections using execution data from other tests (i.e. data that has not been customised to explore this specific effect).

Testing Activity

Revisiting our running example from Section 4.2.2, our aim is to determine the effect of prioritising vaccination for the elderly on the following outputs: cumulative infections, number of doses given, maximum number of doses per agent, and number of agents vaccinated.

Our expectation here is that prioritising the elderly should lead to an increase in infections. This is because we are less likely to vaccinate agents in the model with a greater propensity for spreading the virus (e.g. younger individuals who attend a school or workplace). We also expect the number of vaccines and doses administered to decrease as there are fewer elderly agents in the model. In contrast, the maximum number of doses should not change, as the vaccine is set to be administered at most two times per agent.

Data Generation

We executed the model under two input configurations 30 times each using an experimental data collector (see Section 6.3.3) for every test. For both input configurations, we used the default Covasim parameters, but fixed the simulation length to 50 days, initial infected agents to 1000, population size to 50,000, and made the default Pfizer vaccine available from day seven. However, for the second configuration, we

[4]https://github.com/CITCOM-project/CausalTestingFramework/tree/683e6c55/examples/covasim_/vaccinating_elderly

also sub-targeted (prioritised) vaccination to the elderly using the `vaccinate_by_age` method from the Covasim vaccination tutorial[5].

Causal Testing

Although we provide a causal DAG (Example 4.6) as an illustrative example for this scenario in Section 4.2.2, it is not necessary to perform identification since, under the experimental mode of operation (Section 6.3.3), we explicitly control for potential biases. Consequently, there is no confounding to adjust for in the resulting data, enabling us to calculate the ATE directly by contrasting the average cumulative infections produced by the control (vaccinate everyone) and treatment executions (prioritise the elderly).

Results

As expected, prioritising the elderly causes the cumulative infections to increase (ATE: 2399.7, 95% CIs: [2323.7, 2475.8]) and causes no change to the maximum doses (ATE: 8.9×10^{-16}, 95% CIs: [3.7×10^{-17}, 4.1×10^{-16}]).

However, when we examine the total number of doses given and agents vaccinated (which we would expect to decrease as a result of the more restrictive policy), the tests in fact show that the PUT erroneously causes these outputs to increase sharply by 481351 (95% CIs: [480550, 482152]) and 483506 (95% CIs: [482646, 484367]), respectively. This is an obvious and potentially problematic bug, as it reveals that more agents have been vaccinated than there are agents in the simulation (by a factor of 9.7).

We raised an issue[6] on Covasim's GitHub repository to report this bug in September 2021 and the Covasim developers replied in November confirming that the bug had been fixed for version 3.1. Although the developers did not explain the cause of the bug nor how it was fixed, the change log for version 3.1 stated the following: *Rescaling now does not reset vaccination status; previously, dynamic rescaling erased it.*

This testing scenario has provided insights related to **RQ2** and **RQ3**. Due to employing the experimental mode of the CTF (Section 6.3.3), we have not inferred test outcomes from observational data and therefore this case study does not offer any insights into the accuracy associated with the observational approach.

RQ2 (Efficiency) We used the experimental mode of the CTF to quantify the effect of introducing a vaccination policy on a number of variables, essentially conducting SMT in the conventional way. We repeated both the source and follow-up test cases for each metamorphic relation 30 times for each test (of which there were four), requiring a total of $30 \times 2 \times 4 = 240$ executions of Covasim. We show how, under the experimental mode of operation, the CTF can conduct SMT in the conventional way and demonstrate that, in situations where observational data is unavailable, the CTF can match the efficiency of conventional SMT.

RQ3 (Effort) The amount of human effort required to apply the CTF was low. We did not need to provide a DAG and we did not need to specify a regression model. Instead, the main expenditure of

[5]https://github.com/InstituteforDiseaseModeling/covasim/blob/master/examples/t05_vaccine_subtargeting.py
[6]https://github.com/InstituteforDiseaseModeling/covasim/issues/370

human effort in this case study lies in the process of implementing the test harness for experimental data collection; a step that is required for most model-based testing techniques.

Overall, this case study has demonstrated how the CTF can also be employed under the experimental mode of operation to essentially conduct a conventional SMT approach. This revealed a problematic bug related to vaccination, highlighting the importance of applying metamorphic testing in the computational modelling context.

6.4.3 Covasim: Observational Causal Testing

We now consider the effect of increasing transmissibility (β) on cumulative infections, but this time applying the CTF to simulated confounded observational data. Here, we compare the outcomes inferred by the CTF to the same outcomes achieved using a conventional SMT approach. Our goal here is to understand whether the CTF can operate accurately and efficiently within the challenging context presented by Covasim.

This case study presents a significant testing challenge. There are 156 distinct locations that can be simulated in Covasim that will lead to differing rates of transmission. This is because different locations are modelled with different age distributions and household contact patterns, leading to differences in key attributes of the population, such as susceptibility, that also affect infection dynamics.

Furthermore, Covasim is non-deterministic. Each metamorphic test requires multiple repeats of the source and follow-up tests, making conventional SMT extremely costly in this context. For example, if we repeat both the source and follow-up test cases 30 times for each location, we would need to run $30 \times 2 \times 156 = 9360$ simulations. Although we do not provide precise timing measurements, on a moderate specification machine[7] each of these runs takes between 1 and 2 minutes to complete, requiring between 156 and 312 hours to run all simulations (without parallelisation). The code for this case study can be found in our open source repository[8].

Data Generation

When reasoning about transmissibility and the spread of COVID-19 using Covasim, there are several parameters that can affect the output. These include the variant of the virus and population characteristics such as age and household size, with older populations being more susceptible to infection and higher household contacts leading to quicker viral spread. These population characteristics cannot be specified directly, but can be indirectly altered by selecting a geographical location.

For this case study, we generate two sets of data. First, we directly apply a conventional SMT approach to Covasim in which we execute the model 30 times with $\beta = 0.016$ and $\beta = 0.02672$ for each location, before averaging and contrasting their respective cumulative infections. We select these values of β as they correspond to the β values for the Beta and Alpha variants of COVID-19 available in Covasim.

Second, we simulate (uncontrolled) observational data. To achieve this, we assign a different dominant variant (Alpha, Beta, Delta, Gamma) to each location at random, each of which has its own specific β value ($\beta_\alpha = 0.02672, \beta_\beta = 0.016, \beta_\delta = 0.0352, \beta_\gamma = 0.0328$). For each location, we then create a normal distribution centred around the location-specific β value and a standard deviation of 0.002. We select this standard deviation to give some variance in the exact value of β used for each run of the location,

[7] MacBook Pro, Core i7, 16GB 2133 MHz LPDDR3 RAM

[8] https://github.com/AndrewC19/covasim_case_study/tree/65bc40a

without introducing too much overlap with other variants. We then run 30 simulations for each location, sampling a fresh β value from its distribution on each run. For all simulations, we use a population size of 1 million individuals, 1000 initially infectious individuals, and a duration of 200 days. This results in a data set comprising 4680 simulations (30 per location).

Causal Testing

To begin causal testing, we form our causal specification by specifying a modelling scenario and the causal DAG shown in Figure 6.5. Our modelling scenario uses the default Covasim parameters apart from β (the input-under-study) and the location. We also fixed the duration, population size, and initial infected agents as follows:

$$\{\texttt{days} = 200,\ \texttt{pop\_size} = 1000000,\ \texttt{pop\_infected} = 1000\}$$

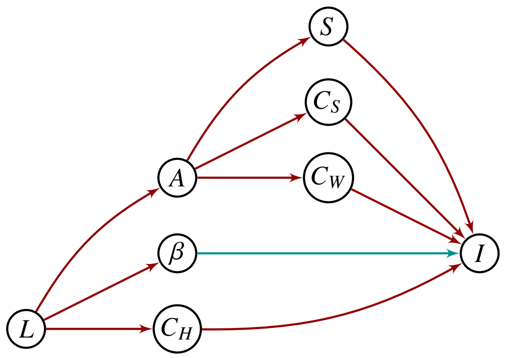

$$\begin{tikzcd}
 & & S \arrow[rdd, bend left] & \\
 & & C_S \arrow[rd] & \\
 & A \arrow[ruu, bend left] \arrow[ru] \arrow[r] & C_W \arrow[r] & I \\
 & \beta \arrow[rru] & & \\
L \arrow[ruu, bend left] \arrow[ru] \arrow[r] & C_H \arrow[rruu, bend right] & &
\end{tikzcd}$$

Figure 6.5: A causal DAG for the Covasim modelling scenario where the causal effect of β on I is confounded. Here, L denotes the location; A denotes the average age of the population; β denotes the transmissibility of the virus; C_H, C_S, C_W denote household, school, and workplace contacts; S denotes average susceptibility of the population; and I denotes the total cumulative infections.

Next, we consider the adjustment sets implied by the causal DAG in Figure 6.5. While there are many possible adjustment sets for this causal DAG, there are three notable choices to discuss.

First, we could use the smallest adjustment set $\{L\}$. This has the advantage of conditioning on the least variables, but restricts estimation to using location-specific data only (i.e. not borrowing data from *similar* locations). Second, we could use $\{A, C_H\}$. This would enable us to additionally borrow information from locations that have similar average ages and household contacts. From an information theoretic standpoint, however, this is not a sensible choice as the average age is not a good measure for the shape of the age distribution (two populations with a similar average age may have vastly different age distributions). To this end, we can consider a third adjustment set $\{S, C_S, C_W, C_H\}$. Here, we replace A with the variables related to age that directly affect cumulative infections: the number of school and workplace contacts (assignment to these environments is determined by age) and susceptibility (which varies with age).

For this case study, we select this third adjustment set on the basis that it most accurately captures the key causal measures while allowing us to borrow data from other locations that are similar with respect

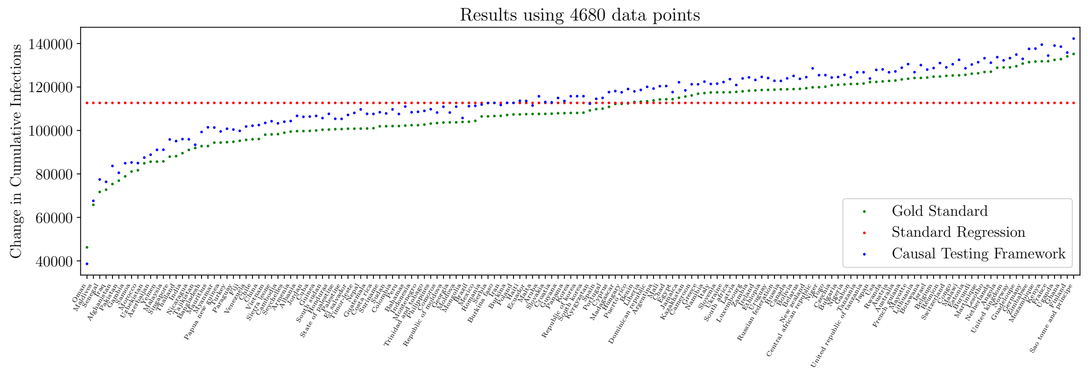


Figure 6.6: A comparison of the metamorphic test outcomes predicted by the CTF and a naive regression model. The metamorphic test in question increases the value of β from 0.016 to 0.02672.

to these attributes. This yields the following closed-form statistical expression that is capable of directly estimating the causal effect (CATE) of interest:

$$CATE = \mathbb{E}[I \mid \beta = 0.02672, S, C_S, C_W, C_H] - \mathbb{E}[I \mid \beta = 0.016, S, C_S, C_W, C_H]$$

Then, to estimate the value of this estimand, we implement a regression model of the following form, where Z is our adjustment set $\{S, C_S, C_W, C_H\}$ and each variable in this adjustment set has three coefficients: x_1^z, x_2^z, x_3^z:

$$I \sim x_0 + x_1 ln(\beta) + \sum_{z \in Z} x_1^z ln(z) + x_2^z ln(z)^2 + x_3^z ln(z) ln(\beta)$$

This regression model encodes three key assumptions. First, due to the exponential nature of viral infection, we apply a log transformation to the variables on the right-hand-side of the equation [179, 180]. Second, we add a quadratic term for each of our adjusted variables. This captures the possibility of curvilinear relationships between I and the parameters. Third, we include an interaction term between β and each of our adjusted parameters. This captures our expectation that the effect of β on cumulative infections is moderated by the number of contacts and susceptibility of the population, and enables the model to make location-specific estimates i.e. CATEs (see Section 2.2.8)[9].

At this point, we have specified a causally-valid statistical model that is capable of directly estimating the causal effect of β on cumulative infections for each location separately. We can therefore compute the average values for the variables S, C_S, C_W, and C_H for each location using our observational data, and substitute these into the model alongside the values $\beta = ln(0.016)$ and $\beta = ln(0.2672)$[10]. By contrasting the respective estimates for I, we obtain an estimate of the causal effect for each location in Covasim.

Results

Figure 6.6 summarises the results of applying the CTF to Covasim to predict the effect of increasing transmissibility (β) on cumulative infections across all locations. These results show three values for

[9]We formed these assumptions by varying each parameter in isolation and observing the change in cumulative infections. An epidemiologist, however, may know more precise characterisations of these relationships a priori.

[10]We take logarithms of the treatment and control values here to maintain the interpretability of our estimate.

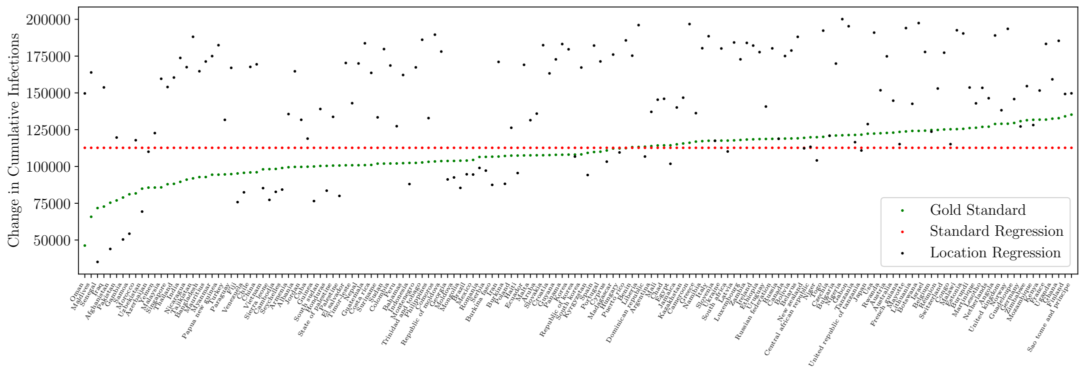


Figure 6.7: A comparison of the metamorphic test outcomes predicted by a naive regression model and the same model with an interaction between location and β.

each location: (i) the gold standard achieved by applying an SMT approach, (ii) a naive estimate with the simple regression model $I \sim x_0 + x_1 ln(\beta) + x_2 ln(\beta)^2$ (i.e. without employing causal knowledge), and (iii) a causal estimate achieved using the CTF and the approach outlined in this section.

By comparing the CTF results to the gold standard shown in Figure 6.6, we can see that the CTF is able to estimate the effect of increasing β from 0.016 to 0.02672 for each location with reasonable accuracy. Specifically, across the location specific estimates, the CTF has a root mean square percentage error ($RMSPE$) of 0.055. This outperforms the naive regression model which provides a uniform prediction that is moderately accurate for 'average' locations, but extremely inaccurate for more 'extreme' locations ($RMSPE = 0.2$).

While these results suggest that the CTF generally overestimates the effect by an average of roughly 5.5% cumulative infections, the overall ordering of the predicted effect sizes is generally consistent with that of the gold standard. We tested this preservation of ordering by calculating the Kendall rank correlation between the (ascending) ordering of the CTF results and the gold standard, returning a value of 0.944 ($p < 0.005$).

By contrast, Figure 6.7 shows the results achieved using the smallest adjustment set, L, and regression model $I \sim x_0 + x_1 ln(\beta) + x_2 ln(\beta)^2 + x_3 L + x_4 ln(\beta)L$. This approach makes location-specific estimates using only the data available for the location in question and is essentially an attempt to apply SMT to incomplete, confounded data. Because each location-specific stratum contains only 30 executions that cover a narrow range of β values, the regression model has to make inaccurate extrapolations, leading to significant over- and under-estimates of the true effect ($RMSPE = 0.515$) and poor rank preservation, as indicated by a Kendall's rank correlation of 0.228 ($p < 0.005$). This stark contrast in performance highlights the value of employing causal knowledge and domain expertise to use data more efficiently.

While Figure 6.6 demonstrates the accuracy with which the CTF can predict SMT outcomes from confounded observational data, these results used the full data set comprising 4680 simulations. Although this is half of the 9360 executions that would typically be required for a conventional SMT approach, this is still a significant amount of data that may not be available in practice. To investigate how much is necessary in practice, we repeatedly applied the CTF to 500 randomly sampled subsets of the data of

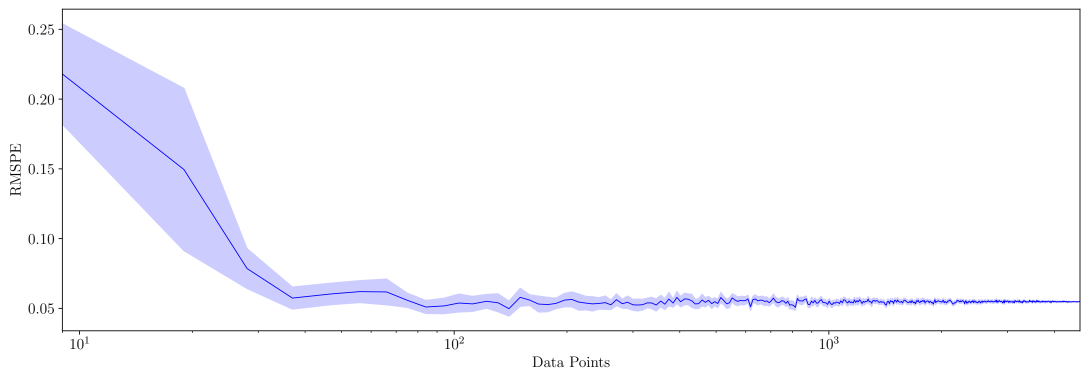


Figure 6.8: Relationship between root mean square percentage error (RMSPE) of CTF predictions and amount of data used (log scale) with 95% confidence intervals.

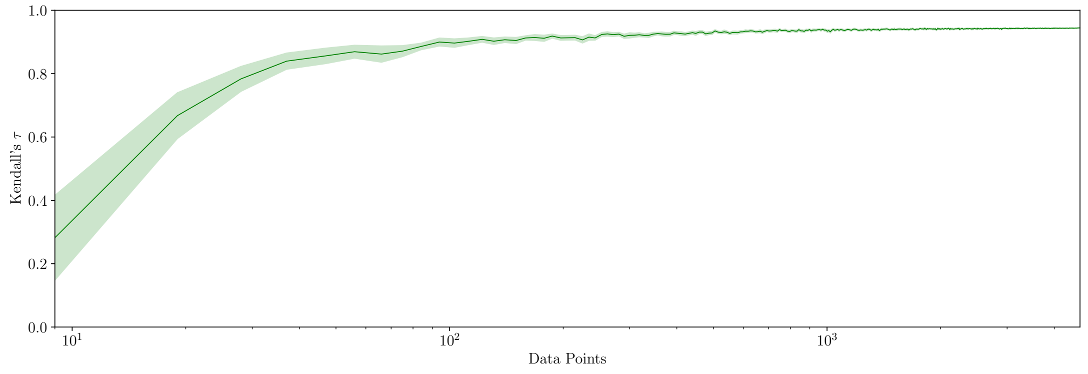


Figure 6.9: Relationship between Kendall's rank correlation (τ) of CTF predictions and amount of data used (log scale) with 95% confidence intervals.

decreasing size and calculated the RMSPE and Kendall's rank correlation. We repeated this process 30 times to obtain a distribution of outcomes and report 95% confidence intervals to demonstrate the error. Figure 6.8 and Figure 6.9 show the results of these experiments. We use a logarithmic scale on the x-axis for these figures as the accuracy changes most significantly between 1 and 200 data points.

Figure 6.8 shows that the RMSPE is greatest with small amounts of data (tens of data points) and quickly reduces to a stable RMSPE of roughly 0.06 by around 200 data points. Similarly, Figure 6.9 shows that the Kendall's rank correlation is initially low (between 0.2 and 0.4) but rapidly increases to a stable value of around 0.9 when 100 to 200 data points are available. This plateau in absolute and comparative error reduction indicates that SMT outcomes can be accurately predicted using only small amounts of data and that larger amounts of data provide negligible gains in accuracy.

This testing scenario has provided evidence for all research questions.

RQ1 (Accuracy) Figure 6.6 shows the accuracy with which the CTF can infer a series of 156 SMT outcomes from confounded observational data *a posteriori*. Although the majority of estimates miss the true effect by around 5.5%, the ordering of the effect sizes is largely consistent with the gold standard. This finding suggests that, in this case study, the CTF is better suited to drawing comparative conclusions about the effect sizes, such as *"Oman is affected significantly less than Finland"* than absolute conclusions, such as *"Finland observes an increase in cumulative infections of 135829"*.

RQ2 (Efficiency) As shown in Figures 6.8 and 6.9, after 200 data points, there is negligible improvement to the absolute and comparative accuracy of the estimator. This suggests that, in this case study, the CTF is significantly more efficient than a conventional SMT approach which would require 9360 executions of the PUT (assuming the source and follow-up tests are repeated 30 times each), with each execution requiring roughly one to two minutes on a moderate specification machine, as noted in earlier in this case study.

RQ3 (Practicality) In this case study, we leveraged our limited domain expertise to specify a causal DAG and regression model that facilitates efficient and accurate inference of test outcomes. Most notably, to borrow data from similar locations, we leveraged our knowledge of viral transmission in Covasim to add terms to our regression model for the attributes that influence the effect of transmissibility on cumulative infections, such as contacts and susceptibility. We achieved this using a relatively small DAG containing only eight nodes and employing commonplace regression modelling techniques, such as quadratic, logarithmic, and interaction terms.

Overall the findings of this case study highlight the potential offered by a causal inference-led approach to SMT: whereas a conventional SMT approach would require thousands of carefully controlled executions to test 156 metamorphic relations, the CTF can accurately infer these outcomes from only 200 data points. Furthermore, the CTF enables a tester to infer these outcomes *a posteriori* from potentially confounded data instead of executing the PUT further times. This approach essentially relaxes the constraints ordinarily placed on data used for SMT, facilitating the re-use of existing data while maintaining the ability to draw *causal conclusions*.

6.4.4 A more advanced regression model for Covasim

In Section 6.4.3, we designed a regression model that broadly captures the expected relationship between cumulative infections and various causally relevant parameters, such as transmissibility β and household contacts C_H. This regression model uses conventional regression modelling techniques to specify the relationships of interest. Namely, quadratic terms, log transformations, and effect modifiers.

However, this model does not capture the relationship between β and cumulative infections perfectly because the relationship follows a sigmoid function (i.e. a characteristic S-shaped curve). Informally, we can explain this relationship as follows. Initially, when β is low, there are few infections because the rate of viral transmission is low. Then, as β increases past some critical threshold, an exponential growth in the transmission rate occurs. Eventually, enough of the population becomes infected and gains immunity or dies, rapidly reducing the rate of viral transmission. This sudden reduction causes cumulative infections to level off, completing the characteristic S shape.

One of the weaknesses of polynomial regression is its unpredictable tail behaviour [80]. This limitation is particularly problematic for modelling sigmoid (S-shaped) relationships, where the tails are necessarily flat. To address this limitation, we employed a more advanced form of regression known as spline regression [82] (Section 2.2.8).

In short, spline regression involves constructing a piece-wise polynomial over contiguous regions of the data. Within each region, a separate polynomial function of degree n is fit to the subset of data. This approach to regression essentially breaks the problem into discrete stages and is an effective technique for capturing non-linear relationships. In many cases, a third-degree polynomial is used to model each region, in which case the resulting splines are referred to as cubic splines (Equation (2.2.20)).

Based on our limited domain expertise, to capture the sigmoid relationship between β and cumulative infections, we used cubic splines with two (internal) knots. With this approach, our aim was to separate the data into three regions corresponding to the three distinct phases of the sigmoid function described above (initial slow growth in infections, exponential growth, and plateau in infections).

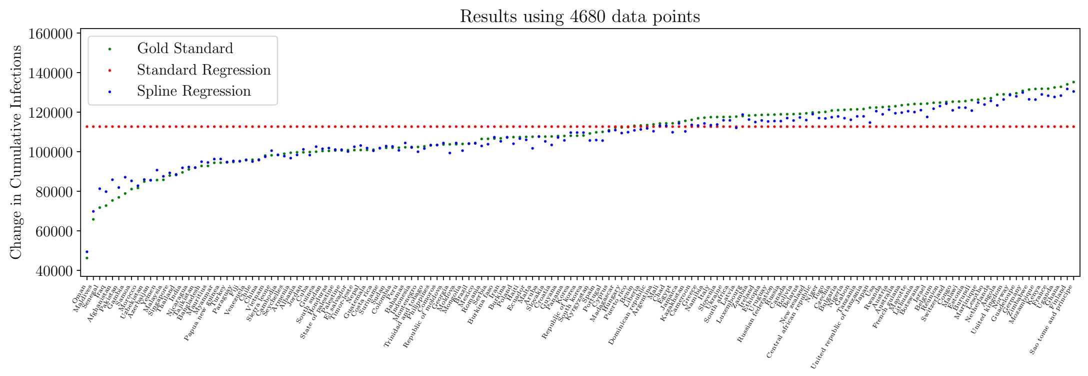


Figure 6.10: A comparison of the metamorphic test outcomes predicted by a cubic spline regression with two knots and the naive regression.

Figure 6.10 shows the metamorphic test outcomes predicted using cubic spline regression. From an informal visual inspection, it is clear that the majority of estimates are more accurate than the previous regression model, which generally overestimated the effects and had a root mean square percentage error ($RMSPE$ of 0.055) and a Kendall's rank correlation of 0.944 ($p < 0.005$). By contrast, the cubic spline approach had an $RMSPE$ of 0.032 and a Kendall's rank correlation of 0.915 ($p < 0.005$). Therefore, the spline regression technique provided better absolute accuracy (indicated by $RMSPE$), but worse comparative accuracy (indicated by Kendall's rank correlation). The performance of both approaches could likely be improved by a domain expert who may have a more precise characterisation of the anticipated relationships.

Overall, we were able to configure the spline regression model in a logical way that is justified by domain expertise (i.e. splitting the relationship into three key regions, each of which can be modelled with a cubic polynomial). This shows how more advanced statistical methods, such as cubic splines, can be employed with arguably less domain knowledge to learn intricate shapes from the available data. However, this introduces an additional burden: the need for expertise in such modelling techniques. In future work, we will investigate the application of other semi- and non-parametric statistical models, such as causal forests [85], within the CTF.

6.5 Discussion

In this section, we discuss the findings of our three research questions outlined in Section 6.4, pertaining to the *accuracy*, *efficiency*, and *practicality* of the proposed approach. We also discuss notable additional findings that fall outside the scope of our research questions.

6.5.1 RQ1 (Accuracy): Can we reproduce the results of a conventional metamorphic testing approach by applying the CTF to observational data?

Throughout our case studies, we applied the CTF to a number of different subject systems from different domains to predict metamorphic test outcomes from observational data. That is, data that had not been collected specifically for the testing task in question.

For example, in Section 6.4.1, we predicted several metamorphic test outcomes for a cardiac action potential model, reproducing the results of an existing study. Moreover, in Section 6.4.3, we then showed how observational data could be re-used to predict multiple different SMT outcomes for an epidemiological model with high comparative accuracy.

The CTF is able to accurately reproduce the results of metamorphic testing across a range of computational modelling software.

This finding suggests that, by leveraging causal inference, the CTF can offer an alternative approach to SMT that does not rely on many potentially costly executions of the SUT. Instead, the CTF can be employed *retrospectively* to infer test outcomes from existing, potentially confounded test data, effectively relaxing the constraints ordinarily imposed on the data used for SMT. In this way, the CTF makes it possible to apply SMT where conventional approaches are currently prohibitively expensive, thereby mitigating the problem of long execution times, as discussed in Chapter 3 and Kanewala and Bieman's survey [13].

6.5.2 RQ2 (Efficiency): In terms of the amount of data required, is the CTF more cost-effective than a conventional metamorphic testing approach?

In Section 6.4.3 (Covasim), we used the CTF to conduct SMT using significantly less data than would be required by a conventional SMT approach. Specifically, we used the CTF to infer the outcomes of 156 distinct metamorphic relations, as shown in Figure 6.6, using roughly half the amount of data required by a conventional SMT approach. We then incrementally reduced the amount of data and repeated our analysis to understand how the accuracy of the approach varies with respect to the amount of data, finding that near-identical results could be achieved using only 200 data points. This corresponds to roughly 2% of the data used by the conventional metamorphic testing approach.

Furthermore, although we have not obtained precise timing measurements, we note that the CTF takes roughly a minute to produce all 156 of the location-specific effect estimates shown in Figure 6.6 on a moderate specification machine. On the other hand, an individual run of Covasim with the settings used in this case study took between one and two minutes on the same machine, and 9360 executions would be required to test these 156 effects using conventional SMT (with 30 repeats per source and follow-up test case). This would amount to between 156 and 312 hours without parallelisation.

The CTF is capable of reproducing the results of SMT using significantly less time and data than is required by a conventional SMT approach.

These findings demonstrate the potential of the CTF to infer the outcomes of metamorphic test cases using significantly less time and data than is required by a conventional SMT approach. Therefore, in conjunction with our findings for **RQ1**, our answer to **RQ2** suggests that the CTF can offer an efficient alternative to conventional metamorphic testing that is more compatible with the notoriously demanding properties of computational modelling software, such as non-deterministic behaviour and long execution times, as described in Chapter 3.

An open question surrounding the efficiency of the CTF is how the quality and diversity of the available data affects the accuracy and scope of inferences. To this end, an interesting avenue for future work would be to investigate how existing test generation and selection strategies can be combined with the CTF to generate and prioritise test cases that, once executed, produce execution data with the greatest inferential power (an informal notion of data adequacy that we briefly discussed in Section 4.2). In a similar vein, Bareinboim and Pearl [168] have proposed general-purpose methods to combine different data sources generated under different conditions to maximise what can be learned from the data. Future work could also investigate how these data fusion techniques can be leveraged in a software testing context to further the inferential power of available data sources.

6.5.3 RQ3 (Practicality): What practical effort is required from the tester to conduct metamorphic testing using the CTF?

Across our case studies, we primarily drew the causal knowledge necessary to elicit the causal DAGs and regression models from existing studies in which the anticipated cause-effect relationships are well-defined. For example, in Section 6.4.1, we used the results of an existing study [172] to specify the causal DAG for the cardiac action potential model (see Figure 6.3). Similarly, in Section 6.4.3, we based the shape of our regression models on basic patterns of infectious disease that could be garnered from the Covasim documentation [128], research paper [127], and our existing experience of the software itself. The main expenditure of human effort here was gathering the domain expertise for each system; converting these into causal DAGs was straightforward and required little time. It stands to reason that this would be less time-consuming for a scientific modeller (for example), who would already have a reasonably strong understanding of the underlying subject matter.

As with any model-based testing technique, time and effort are necessary to obtain knowledge and turn it into a domain model. In addition, this process often assumes familiarity with software-specific notions, such as how to characterise a state in a state machine [181], or what events should (or should not) be possible at any given point. Furthermore, the resulting models tend to contain implementation-specific details likely to be unfamiliar to most scientific software users [13]. By contrast, the CTF relies on an intuitive, domain-agnostic model (i.e. a causal DAG) that makes essential assumptions transparent and requires a basic understanding of regression modelling. This set of requirements poses a lower barrier to entry for a typical user of scientific software.

More generally, from specification to testing, the components of the CTF defined in Section 4.2 assume no prior knowledge of the implementation of the PUT. Instead, the CTF requires the user to specify domain-specific details that are independent of the implementation. This shifts the nature of the burden placed on computational modellers from being software-specific to domain-specific. In doing so, the CTF facilitates the application of state-of-the-art testing techniques, such as metamorphic testing, to computational modelling software *without the user even necessarily knowing what a metamorphic relation or test is*. This has been demonstrated throughout the case studies.

The main expenditure of effort in applying the CTF is the gathering of domain expertise; the task of expressing knowledge in a causal DAG and regression model is comparatively straightforward and involves limited effort. Furthermore, compared to other model-based testing techniques, the barrier to entry for the CTF is better suited to the typical skill set of computational model users.

Our work is based on the contention that the effort required to employ the CTF is not unreasonable and that, relative to most model-based testing techniques, the necessary expertise are more familiar to a typical computational model user [13]. Namely, the ability to elicit anticipated cause-effect relations in a causal DAG and familiarity with regression modelling techniques. However, to precisely quantify and empirically evaluate the feasibility and practicality of the approach, future work will look to conduct a human study in which various computational modelling experts apply the CTF to a range of computational modelling software.

6.5.4 Summary

Collectively, our answers to **RQ1** and **RQ2** suggest that the CTF offers an accurate and efficient approach that addresses several of the challenges associated with the testing of computational modelling software outlined by Kanewala and Bieman [13]. Most notably, through the ability to infer metamorphic test outcomes from small amounts of existing observational data, the CTF mitigates the prohibitively long execution times that typically prevent adequate testing of computational modelling software. Consequently, the CTF also increases the applicability of metamorphic testing to computational modelling software, helping to indirectly alleviate the test oracle problem [182]

Of course, the accuracy and efficiency offered by the CTF come at a cost. Our answer to **RQ3** suggests that the CTF presents a trade-off between practical effort and accuracy/efficiency: by leveraging causal knowledge and domain expertise, the CTF can apply SMT in situations where it is currently impractical. However, these domain expertise can be difficult to obtain for non-domain experts. In the case studies, we found the main expenditure of human effort to be in collecting the domain expertise necessary to apply the techniques; the process of converting these into a causal DAG and regression model required considerably less effort.

6.5.5 Additional Findings

Throughout our case studies, we also identified a number of additional findings that warrant discussion. First, we discuss the need for explainability and how causal DAGs help to address this. Second, we discuss a noteworthy bug identified using the CTF.

Explainability When testing computational modelling software, the reasoning behind a particular test passing or failing (i.e. the test oracle procedure) is rarely made explicit. For example, modellers often use regression testing to check whether changes to the PUT have affected model predictions or results. Any deviations are then typically validated by a domain expert. This form of ad-hoc validation lacks transparency and, as such, cannot be easily interrogated by prospective users of the PUT. For applications such as infectious disease modelling, where software outputs may inform important policy decisions, there is a need for accountable and explainable test results. Explainability is also a topic of growing concern in fields such as healthcare [183] that are increasingly using black-box machine learning techniques but require transparent, accountable, and interpretable decision making [184].

To this end, the CTF incorporates *explainability* into the testing process. Specifically, by utilising causal DAGs for causal inference, the CTF includes a lightweight and transparent artefact that partially explains the reasoning behind reaching a particular test outcome (i.e. why a specific adjustment set, and therefore statistical model, yields a *causal* estimate). Furthermore, the causal test case (Definition 4.5) includes an explicit test oracle (Definition 4.7) that captures ‘correctness’ in terms of some causal metric, such as the ATE or RR. Both assets can be easily accessed and interrogated, increasing the explainability and reputability of tests.

With this built-in notion of explainability, we posit that the CTF also has the potential to complement existing techniques in the computational modelling context that often rely on implicit domain expertise for testing, such as regression testing. However, the causal DAG and test oracle do not communicate all assumptions with the potential to influence test results and their interpretation. For example, the anticipated functional form of a particular cause-effect relationship will influence the design of the regression model and its resulting predictions. A potential avenue for future work would be to investigate methods for improving the explainability of the CTF. For example, one could look into more expressive graphical causal models that also capture the expected functional form.

Covasim Bug Our case studies also revealed an interesting, previously undiscovered bug in Covasim’s vaccine implementation. Specifically, upon prioritising the elderly for vaccination, the number of vaccinated individuals grew to nearly ten times the number of individuals in the simulation. While this does not appear to significantly affect the key outputs of the model, it is not difficult to imagine how such a bug could lead to an overestimation of the effects of interventions.

6.5.6 Threats to Validity

Our evaluative case studies in Section 6.4 do not claim to make generalisable conclusions regarding the accuracy, efficiency, and effort associated with the CTF. Instead, these case studies serve as proofs of concept that show - for the studied subject systems - how formulating metamorphic testing as a causal inference problem makes it possible to apply the approach in situations where conventional metamorphic testing methods are impractical. Nonetheless, there are some threats to validity worth considering here.

External Validity

In this chapter, the main threat to external validity is that our case studies only cover two subject systems involving a moderate number of input and output variables. As graphical causal inference requires domain expertise for the data-generating mechanism in the form of a causal DAG, a significant amount of time was spent familiarising ourselves with the subject systems and understanding their constituent cause-effect relationships. As a result, this limited our ability to systematically collect and analyse large numbers of varied subject systems.

Furthermore, our subject systems were all implemented in Python. Therefore, our findings do not necessarily generalise to computational modelling software implemented in other languages. However, the CTF only requires execution data in CSV format to perform causal testing observationally and can thus be applied, in theory, to tabular data produced by *any* scientific model.

As a consequence of the aforementioned threats to external validity, we acknowledge that our results may not generalise to *all* forms of computational modelling software. However, we attempt to mitigate the aforementioned threats to external validity by selecting models that differ in their complexity, subject matter, and modelling paradigm. In addition, as discussed in Section 6.4, the selected systems have important but vastly different applications in different domains, and have both been the subject of prior research.

Internal Validity

In this chapter, the main threat to internal validity is that we did not optimise the estimators and configuration parameters thereof for our case studies. While this avoids the problem of over-fitting, it means there may exist statistical models that are more suitable for modelling and inferring the behaviour of the input-output relationships under study.

In the same vein, we specified regression equations that capture the expected functional form of various input-output relationships. For example, when testing Covasim in Section 6.4.3, we specified a regression model which captures our broad understanding of how cumulative infections vary with various causally relevant parameters. We called upon our experience with the models and subject area to specify these equations. However, different domain experts may have different opinions about the correct functional forms of the input-out relationships and may therefore have specified these relationships differently or more accurately.

As a consequence of the above threats to internal validity, we acknowledge that alternative statistical models may achieve more precise causal inferences for the subject systems. However, we partially mitigate the above threats to internal validity by manually inspecting the functional forms of the relationships between inputs and outputs of interest in the PUT. We achieve this by varying one parameter at a time and observing how the output in question changes in response (in a similar way to our case study in

Section 6.4.1). We also applied a more advanced regression model in Section 6.4.4 that more accurately captures the relationship between transmissibility (β) and the number of cumulative infections (I) in Covasim.

6.6 Concluding Remarks

In this chapter, we provided a reference implementation of the CTF; our conceptual framework from Chapter 4 that facilitates the application of causal inference techniques to software testing problems. This framework follows a model-based testing approach to incorporate an explicit model of causality into the software testing process in the form of a causal DAG, enabling the direct application of graphical causal inference methods to software testing activities.

Motivated by the often high costs associated with performing metamorphic testing on computational modelling software (as demonstrated in Chapter 3), in this chapter, we demonstrated how the cost of metamorphic testing can be reduced when viewed and solved as a problem of causal inference. In particular, through our reference implementation of the CTF, we used graphical causal inference methods and regression modelling techniques to predict metamorphic test outcomes from existing, confounded execution data.

To further understand the accuracy, efficiency, and practicality of the proposed approach, we applied our open source reference implementation of the CTF to real-world computational models of varying size and complexity, including a cardiac action potential model and the epidemiological ABM that formed our second motivating example in Chapter 3. The results of these case studies suggest that, through the use of causal inference, the CTF can accurately infer metamorphic test outcomes for these computational models from existing test data using significantly less data than is required by a conventional SMT approach. This evidence indicates the potential of causal inference-led testing methods to provide a cost effective alternative for testing computational modelling software in situations where it is currently impractical or infeasible.

While this chapter has demonstrated how causal inference can reduce the cost associated with metamorphic testing of computational modelling software, there are a number of avenues through which the proposed approach may be improved and several alternative research directions to be considered.

First, metamorphic testing is only one example of a causal testing activity. Other testing activities with a fundamentally causal objective should also be explored to understand how causal inference could be of benefit. For example, regression testing typically involves detecting whether a particular change to some software causes any previously passing tests to subsequently fail. This activity clearly shares the same fundamental components as other problems of causal inference - namely: an intervention in the form of a software change and an outcome in the form of a test regression (i.e. pass to fail). Therefore, it has the potential to be translated to, and perhaps benefit from, being solved as a problem of causal inference in future work.

Second, although the results of the case studies (Section 6.4) indicate that the proposed approach can significantly decrease the amount of data needed to perform metamorphic testing (and therefore execution costs involved), they also highlight the additional overhead of using graphical approaches to causal inference. In particular, the need to specify a causal DAG and regression model will inevitably require time and effort. To reduce the cost associated with providing these artefacts, future work could explore automatic approaches to the generation of causal DAGs, such as those provided by the growing field of causal discovery [159], and the use of more advanced non-parametric methods to estimation, such as those briefly discussed in Section 2.2.9.

Overall, in this chapter, we have:

- Provided an open source Python reference implementation of the CTF that facilitates the application of causal inference methods to causality-focused software testing tasks.
- Proposed an alternative method for conducting metamorphic testing that uses graphical causal inference methods and regression modelling techniques to predict metamorphic test outcomes from observational data.
- Conducted a series of case studies involving real-world computational models in which we evaluated the accuracy, cost, and effort associated with the aforementioned approach.
- Presented results showing that observational causal inference can significantly reduce the amount of data needed to perform metamorphic testing, with the caveat that additional effort is needed to specify a causal DAG and regression model.

Following this chapter, we conclude this thesis by reflecting on the contributions made in each chapter. We then outline several avenues for future work that are related to the themes explored in this thesis, before delivering our final remarks.

7 | Conclusions

Computational models are notoriously difficult yet important to test. In this context, a bug or faulty modelling assumption could have catastrophic consequences at an unprecedented scale, including extreme economic damage and even preventable loss of life. Despite the evident importance of this class of software, systematic testing is seldom applied in practice as existing techniques do not account for the challenging characteristics posed by computational models.

Most notably, computational models are known to suffer from the test oracle problem [15]: How can a modeller possibly tell if the model is behaving correctly, given that they are simulating events that have not yet occurred? To address this problem, modellers may turn to metamorphic testing, an approach that effectively mitigates the test oracle problem by performing specific *changes* to the input space and checking whether they have the expected *effect* on the output space.

However, computational models typically exhibit a number of challenging characteristics, such as long execution times and non-determinism, that pose a significant barrier to systematic metamorphic testing. In this thesis, we have made steps towards breaking down this barrier and increasing the applicability of metamorphic testing to computational modelling software.

To achieve this, we proposed and evaluated the use of causal inference methods based on two observations: First, many software testing tasks, including metamorphic testing, are fundamentally causal in nature and share parallels with methods of causal inference. Second, in other fields, causal inference has been used to overcome limitations similar to those that hinder the testing of computational models (inability to run experiments/tests due to physical or practical constraints).

We now summarise and critically analyse the key findings of the core chapters of this thesis.

7.1 Summary of Contributions

In Section 1.1, we outlined two primary objectives that this thesis aims to achieve:

RO1 To explore how, *in theory*, causal inference can increase the applicability of metamorphic testing to computational modelling software.

RO2 To explore whether, *in practice*, causal inference can increase the applicability of metamorphic testing to computational modelling software.

We now outline the contributions made in each of the core chapters of this thesis towards the aforementioned research objectives.

7.1.1 Metamorphic Testing of Computational Modelling Software

In Chapter 3, we analysed and applied metamorphic testing to two real-world epidemiological models. Initially, we focused on the simpler and theoretically easier to test EBM of the 1918 Spanish Influenza before moving onto the significantly more advanced Covasim ABM. Using these examples, we outlined numerous characteristics that pose practical difficulties to testing such software, including:

1. Long execution times
2. Uncertainty in inputs and outputs
3. Vast input and output space
4. Non-determinism

In addition, we highlighted a more fundamental issue that is synonymous with computational models such as these: *the test oracle problem* [15].

With this in mind, we then applied metamorphic testing to both motivating examples to demonstrate how it can alleviate the test oracle problem. Here, we re-implemented a series of metamorphic relations from previous work [5] that were designed for epidemiological modelling software, and monitored the execution time of the approach. This led to several interesting findings.

First, we found that some of the metamorphic relations outlined in previous work did not hold in all executions. More specifically, when the input parameters of certain metamorphic relations were varied by too big or too small of a factor, the outputs did not change in the expected way, causing the metamorphic relations to fail. This can be attributed to an imprecise specification of the input boundaries and highlights the difficulty in designing suitable tests for such complex software.

Second, we found that the cost of metamorphic testing grows significantly when applied to the more advanced ABM. This is not only a result of the longer individual execution times, but is compounded by the non-deterministic nature of the ABM which necessitates the use of SMT. This more advanced form of metamorphic testing essentially requires every metamorphic test to be repeated numerous times.

Overall, in Chapter 3, we made the following contributions:

- We applied metamorphic testing to two real-world epidemiological models, re-implementing a series of metamorphic relations designed for epidemiological software.
- We highlighted several practical limitations of metamorphic testing for this class of software, as well as weaknesses in the metamorphic relations proposed in previous work.

With these contributions in mind, we now answer to **RQ1.1** from Section 1.1 as follows:

RQ1.1: *What are the main factors that hinder the application of metamorphic testing to computational modelling software?*

There are two main factors that currently hinder the application of metamorphic testing to computational models. First, metamorphic relations are notoriously difficult and time consuming to construct. This issue is exacerbated by the challenging subject matter typically simulated by computational models (e.g. viral transmission in a pandemic), meaning that specialist knowledge is often needed to devise metamorphic relations. Second, the significant computational overhead of this class of software coupled with its often non-deterministic nature make metamorphic testing costly and thus often impractical.

7.1.2 Causal Testing Framework

Motivated by the issues of practicality surrounding metamorphic testing outlined in the Chapter 3, in Chapter 4, we then discussed the proposed solution to be explored in this thesis: A causal inference-driven software testing framework.

Initially, to justify this direction, we discussed how causal inference methods have been employed in fields like epidemiology to establish and measure causal relationships where experimental procedures are unethical or otherwise infeasible. We then argued that metamorphic testing has essentially the same goal — to establish the causal effect of some intervention — and can therefore be expressed and solved as a problem of causal inference.

With this motivation in mind, we then extended the conceptual STF introduced by Staats et al. [1] to incorporate the artefacts needed to facilitate the application of causal inference methods to software testing problems. As in the original paper [1], we used this framework to discuss the important interactions between its components and, in addition, highlighted the nuances that distinguish our CTF from the conventional STF.

As an example, we then demonstrated how metamorphic testing can be framed as a causal inference problem within this framework and outlined four potential advantages of this formulation:

- *Transportability*: The ability to directly apply a wealth of existing causal inference methods to metamorphic testing.
- *Cost effective test executions*: The potential to perform metamorphic testing retrospectively using observational data, where conventional test executions are prohibitively expensive.
- *Universal and testable properties*: The ability to leverage graphical causal models as a means for identifying a series of testable causal properties that are not domain specific (i.e. primitive metamorphic relations).
- *Transparency*: The key assumptions upon which the validity of any causal test outcome depends are made contestable and transparent (e.g. causal DAG for causal assumptions; modelling scenario for relevance of data; causal test oracle for the procedure followed to ascertain the validity of test outcomes).

Overall, in Chapter 4, we made the following contributions:

- We extended the conceptual STF proposed by Staats et al. to include the fundamental causal artefacts needed to facilitate the application of causal inference methods to software testing problems.
- Given its fundamentally causal objective, we demonstrated how metamorphic testing can be translated to a problem of causal inference through this framework and outlined the potential advantages of this formulation.

With the aforementioned contributions in mind, we can now provide answers to **RQ1.2** and **RQ1.3** from Section 1.1:

RQ1.2: *How can we introduce causal inference to a software testing framework?*

To facilitate the application of causal inference to causality-focused software problems, we extended the STF of Staats et al. [1] to form the CTF. In particular, this involved defining causal versions of test cases, which outline the expected *causal effect* of an *intervention* within a particular *modelling scenario*, and causal test oracles, which determine the correctness of a causal test case by examining *causal effects*. In addition, we introduced data as its own first class entity, given its importance in causal inference.

RQ1.3: *What are the potential advantages to formulating metamorphic testing as a problem of causal inference?*

We outlined four potential advantages of approaching metamorphic testing as a causal inference problem: transportability of existing causal inference methods, cost effective test executions, universal and testable properties, and transparency of key assumptions.

At this point, it is worth reflecting on **RO1**. In Chapter 3, we provided empirical evidence that demonstrates the high cost associated with metamorphic testing when applied to computational models with particular characteristics. It stands to reason that such practical issues pose a significant barrier to the systematic testing of computational models. To this end, in Chapter 4, we explored the conceptual overlap between causal inference and metamorphic testing, ultimately demonstrating how metamorphic testing can be expressed as a problem of causal inference and, in doing so, how this may improve the applicability of metamorphic testing to computational models. In Part I of this thesis, we have therefore established how causal inference can, *in theory*, help to improve the applicability of metamorphic testing to computational models.

7.1.3 Identification of Metamorphic Relations from Causal Graphs

In Chapter 5, we proceeded to Part II of this thesis and began to explore how the theoretical advantages of a causal inference-led approach to metamorphic testing could be realised in practice. We started by focusing on one of the well known and arguably most significant barriers to developing metamorphic tests: the process of constructing metamorphic relations. To this end, we proposed the use of causal DAGs (see Definition 2.2) as a form of 'causal specification' that captures the key cause-effect relationships in software (referred to as a Causal Software Graph). We then outlined an algorithm that can be used to generate primitive metamorphic relations and accompanying tests based on its structure.

Through a series of experiments, we demonstrated how this approach can be used to systematically identify metamorphic relations capable of identifying 'structural causal bugs'. We coined this term to refer to defects that affect the underlying causal structure of a program. This form of bug appears to be prevalent in software systems, with a secondary analysis of Sobreia et al.'s study [138] of patches to the Defects4J framework [17] finding that 57.7% of repair actions implied a change in causal structure [139]. Our results suggested that the performance of the approach is robust to misspecifications in the underlying causal DAG (i.e. missing or spurious edges) and can detect evasive causal relationships (i.e. where the causal relationship is only exercised by specific changes in input) when paired with an appropriate test generation strategy. In addition, we demonstrated how the approach could be applied to detect a pair of real-world structural causal bugs from the Defects4J framework [17].

This presents a significant step forward as existing approaches to the generation or construction of metamorphic relations are generally domain specific or rely on predicting whether existing metamorphic relations hold in the PUT. We reviewed these techniques in detail in Section 2.1.5.

We believe this step forward is partly the result of using a *causal* artefact as the oracle data (i.e. the data used by the oracle to ascertain correctness of the PUT) for our approach. This is necessary since a metamorphic relation is a necessarily causal construct - it communicates the expected *change* in output that should be *caused* by a particular *change* in input. This is a potentially interesting finding as it indicates that causal models, such as causal DAGs, are suitable constructs for modelling metamorphic relations.

Overall, in Chapter 5, we made the following contributions:

- We outlined a causal model-based technique for systematically generating primitive metamorphic relations that can detect bugs affecting the causal structure of a program.
- We conducted an empirical evaluation in which we applied the approach to synthetic programs with a known causal structure to detect injected faults that mutate the program's causal structure, finding that the approach is robust to misspecification and capable of detecting conditional causal behaviour.
- We demonstrated how the approach can be applied to actual software to detect a pair of bugs that affect causal structure.

This chapter provides an answer to RQ2.1 from Section 1.1:

RQ2.1: *How can we use causal inference methods to help testers to construct metamorphic relations?*

Provided we can capture the expected cause-effect relationships between key variables involved in a program in a causal DAG, we can automatically analyse its structure to obtain a series of primitive metamorphic relations that are capable of detecting structural causal bugs.

7.1.4 Metamorphic Testing with Observational Data

In Chapter 6, we turned our attention to overcoming a further barrier to applying metamorphic testing that is particularly problematic for computational models: the cost of test execution. Motivated by the parallel objectives of metamorphic testing and causal inference, in this chapter, we investigated whether metamorphic test outcomes could be predicted from existing test data.

To achieve this, we started by providing a reference implementation for the conceptual CTF introduced in Chapter 4. Through a series of case studies focusing on two separate real-world computational models, we then demonstrated how metamorphic testing could be achieved using the CTF both experimentally (i.e. under the conditions necessary to isolate the causal relationship of interest) and observationally (i.e. by predicting the change in output caused by the change in input from observational data).

Moreover, focusing on the observational approach to causal inference, we evaluated the accuracy, cost, and effort needed to employ the approach. We found that we could predict metamorphic test outcomes with reasonable accuracy and using less data than required by a conventional metamorphic testing approach. In particular, when applied to the COVID-19 ABM from Chapter 4, we found that the accuracy of our estimator plateaued at around 200 data points, meaning any further data resulted in minuscule accuracy gains. It is worth noting that, relative to the conventional metamorphic testing approach employed in the study, this represents roughly a 98% decrease in the required data.

To evaluate the effort required, we provided a qualitative account of the main steps involved in applying the approach. We found that the main expenditure of time and effort was in collecting the domain expertise needed to specify the 'shape' of the regression model used for estimation. That is, understanding the subject being simulated with sufficient detail to construct a statistical model that makes sensible modelling assumptions. In addition, the process of modelling these assumptions required a working understanding of regression modelling techniques. On the other hand, the process of constructing the causal DAG and applying the framework itself required little time. Therefore, we concluded that the effort involved is largely dependent on the domain knowledge of the user. However, given that the target user is a modeller or scientist, we maintain that the steps requiring most effort (gathering of domain expertise) are more likely to be familiar and thus unlikely to present a significant barrier to entry.

Overall, in Chapter 6, we made the following contributions:

- We provided a reference implementation of the CTF: a causal inference-driven software testing framework that was previously outlined (as a concept) in Chapter 4.
- We conducted a series of case studies on real-world computational models in which we evaluated the accuracy, cost, and effort necessary to apply the approach.
- We presented empirical evidence demonstrating that the cost of metamorphic testing (i.e. amount of data needed) can be significantly reduced by applying causal inference techniques to observational data, provided a causal DAG and well-specified regression model can be specified.

This chapter enables us to answer RQ2.2 from Section 1.1 as follows:

RQ2.2: *How can we use causal inference methods to reduce the cost associated with executing metamorphic tests, particularly when dealing with costly non-deterministic systems?*

Using the CTF, we can leverage graphical causal inference methods and well-specified regression models to predict metamorphic test outcomes from relatively small amounts of observational data with reasonable accuracy. This has the potential to reduce dependence on costly test executions as approximate metamorphic test outcomes can be inferred from existing data instead.

In regards to **RO2**, Chapters 5 and 6 have demonstrated how causal inference methods can be leveraged to alleviate two problems associated with metamorphic testing: metamorphic relation construction and cost of execution. This builds on the conceptual overlap between causal inference and metamorphic testing that was outlined in Part I of this thesis and gives credence to the theoretical advantages of employing causal inference within the archetypal software testing framework as outlined in Chapter 4. Therefore, in Part II of this thesis, we have established how causal inference can, *in practice*, help to improve the applicability of metamorphic testing to computational models.

7.2 Future Work

In this thesis, we have established a firm relationship between causal inference and metamorphic testing. While we have provided empirical evidence that demonstrates how this relationship can be exploited to improve the applicability of metamorphic testing to computational models, a number of open problems remain.

Expanding the Relationship Between Testing and Causality

In this thesis, we have focused on metamorphic testing as a testing activity. Other testing activities, such as regression testing, also share a similarly *causal* objective. It therefore raises the question: what other testing activities could benefit from being framed as a problem of causal inference. In a similar vein, this thesis has focused on a relatively narrow band of causal inference techniques. A similar open question is whether alternative causal inference methods can also be leveraged in a testing context to some benefit. For example, so-called instrumental variable methods exist that have an entirely different set of identifiability conditions (instead of those introduced in Section 2.2.3) that make them applicable under different circumstances. In particular, these methods can be used where there are known unmeasured confounders [145]. More generally, future work could expand on the relationship between causality and software testing that is investigated in this thesis, by either leveraging different causal inference methods to realise distinct advantages or identifying other testing activities with a fundamentally causal objective.

Generating Metamorphic Relations for Functional Causal Bugs

In Chapter 5, we proposed an approach for systematically constructing metamorphic relations from a causal DAG. Through a series of experiments, we evaluated the ability of the approach to detect *structural causal bugs*. That is, a bug that adds or deletes an edge from the actual causal structure of the PUT (e.g. the result of referencing the wrong variable name). Consequently, the resulting metamorphic relations act as oracles that are limited to testing the presence and absence of expected cause-effect relationships. Another way in which a causal relationship can exhibit faulty behaviour is if its functional form is incorrectly implemented. For example, if an output should vary quadratically with respect to its input, but it actually varies linearly. In this case, there is still a (faulty) causal effect and therefore our approach would not detect the bug. We refer to this form of bug as a *functional causal bug*. Future work could explore using more expressive forms of causal model that additionally express the expected functional form of a relationship. While this would require significantly more input from the user, it could be used to systematically construct more refined metamorphic relations capable of detecting functional causal bugs.

Data Adequacy for Causal Inference-led Testing

In Chapter 6, we used causal inference methods to predict metamorphic test outcomes from small amounts of simulated observational data. While this data had a number of problems, such as missing values (positivity violations; see Section 2.2.3), it was relatively controlled in the sense that several key parameters, such as population size and initial infectious population size, were constant. Intuitively, the less controlled the data is, the more difficult it becomes to draw accurate causal inferences. To combat this, we integrated the notion of a modelling scenario into the CTF that enables the user to place constraints on the observational data that define the scenario of interest. However, an open question is how the accuracy of the approach varies as the modelling scenario (and therefore specificity of the data) is relaxed. More generally, future work could investigate the properties or features of observational data that make it more or less suitable for making causal inferences within the context of testing activities. In this way, there is a need for some form of data adequacy metric that measures how suitable a particular set of data is for the task at hand, in a similar way to test adequacy measuring how 'good' a test suite is for a particular testing objective.

Causality-Guided Test Case Generation

In Chapter 6, we assumed the presence of some set of existing test executions that we could use as observational data. However, an open question is whether causal inference can still be of value where there is no such data. One possibility that could be explored is whether, given some set of causal test objectives that would require n executions (e.g. a set of metamorphic tests that we want to run), is there some set of $m < n$ executions that would enable us to accurately predict all n outcomes? This essentially flips the problem on its head from re-using existing *observational* data to generating a minimal set of new *interventional* data from which accurate estimates can be made. This could be guided by the previously mentioned 'data adequacy metric'.

7.3 Final Remarks

Software testing is an inherently causal task. When a developer tests software and discovers an error, the inherent assumption is that it is *caused* by some defect. Similarly, when designing test cases to expose such defects, a tester typically works on the assumption that the test inputs *cause* the outputs to take on a particular value. Despite its foundational causal nature, causality is a subject that has received relatively little attention in the field of software testing.

In this thesis, we have disrupted this trend and have explicated the causal roots of metamorphic testing. Motivated by the importance of testing computational modelling software, we have established a relationship between causal inference and metamorphic testing, and developed a testing framework that facilitates the application of the former to the latter. Through a series of experiments, we have demonstrated how this relationship can be exploited to overcome existing barriers to metamorphic testing, including the task of constructing metamorphic relations and the cost associated with executing metamorphic test cases. In doing so, we have shown how causal inference methods have the potential to improve the applicability of metamorphic testing to computational models, and therefore made a step towards increasing the testability of this all important class of software.

The relationship between causality and software testing holds much potential and warrants further investigation. We have demonstrated how this potential can be realised in the context of one particular testing activity — metamorphic testing — to the benefit of one particular class of software — computational models. Future work should seek to explore this relationship in greater detail and leverage its benefits for other causality-focused testing activities. Moreover, future work should further establish the effectiveness and practical utility of the proposed causal inference-led approach to metamorphic testing. In particular, understanding what makes data adequate for causal inference in testing settings and how one can most efficiently use data for testing are important questions that should be explored.

Bibliography

[1] M. Staats, M. W. Whalen, and M. P. Heimdahl, "Programs, tests, and oracles: the foundations of testing revisited," in *Proceedings of the 33rd international conference on software engineering*, pp. 391–400, 2011.

[2] B. Neal, "Introduction to causal inference," 2015.

[3] S. R. Sukumar and J. J. Nutaro, "Agent-based vs. equation-based epidemiological models: A model selection case study," in *2012 ASE/IEEE International Conference on BioMedical Computing (BioMedCom)*, pp. 74–79, IEEE, 2012.

[4] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[5] L. L. Pullum and O. Ozmen, "Early results from metamorphic testing of epidemiological models," in *2012 ASE/IEEE International Conference on BioMedical Computing (BioMedCom)*, pp. 62–67, IEEE, 2012.

[6] I. for Government, "Timeline of uk government coronavirus lockdowns and restrictions," 2021.

[7] E. Scott, "Lockdown 1.0 and the pandemic one year on: what do we know about the impacts," *UK Parliament-House of Lords Library*, 2021.

[8] P. Brien and M. Keep, "Public spending during the covid-19 pandemic," *House of Commons Library*, 2023.

[9] B. Nabarro, "Uk economic outlook: the long road to recovery," *The Institute for Fiscal Studies*, 2020.

[10] N. Ferguson, D. Laydon, G. Nedjati Gilani, N. Imai, K. Ainslie, M. Baguelin, S. Bhatia, A. Boonyasiri, Z. Cucunuba Perez, G. Cuomo-Dannenburg, *et al.*, "Report 9: Impact of non-pharmaceutical interventions (npis) to reduce covid19 mortality and healthcare demand," 2020.

[11] D. Adam, "Special report: The simulations driving the world's response to covid-19.," *Nature*, vol. 580, no. 7802, pp. 316–319, 2020.

[12] B. Wynne and G. Ackland, "The long term epidemic predictions from imperial college covidsim report 9," *Age*, vol. 5, p. 10, 2020.

[13] U. Kanewala and J. M. Bieman, "Testing scientific software: A systematic literature review," *Information and software technology*, vol. 56, no. 10, pp. 1219–1232, 2014.

[14] B. C. Society, “Professionalising software development in scientific research,” 2020.

[15] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[16] T. Y. Chen, S. C. Cheung, and S. M. Yiu, “Metamorphic testing: A new approach for generating next test cases,” Tech. Rep. HKUST-CS98-01, The Hong Kong University of Science and Technology, 1998.

[17] R. Just, D. Jalali, and M. D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, (New York, NY, USA), p. 437–440, Association for Computing Machinery, 2014.

[18] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The Art of Software Testing*, vol. 2. Wiley Online Library, 2004.

[19] E. F. Moore, “Gedanken-experiments on sequential machines,” in *Automata Studies. (AM-34)*, pp. 129–154, Princeton University Press, 1956.

[20] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Elsevier, 2010.

[21] H. Zhu, P. A. Hall, and J. H. May, “Software unit test coverage and adequacy,” *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

[22] J. Offutt and A. Abdurazik, “Generating tests from uml specifications,” *«UML»’99—The Unified Modeling Language*, pp. 76–76, 1999.

[23] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, “Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation,” *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 23–42, 2013.

[24] B. J. Grün, D. Schuler, and A. Zeller, “The impact of equivalent mutants,” in *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pp. 192–199, IEEE, 2009.

[25] Y. Jia and M. Harman, “Higher order mutation testing,” *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[26] R. Hierons, M. Harman, and S. Danicic, “Using program slicing to assist in the detection of equivalent mutants,” *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 233–262, 1999.

[27] T. A. Budd and A. S. Gopal, “Program testing by specification mutation,” *Computer languages*, vol. 10, no. 1, pp. 63–73, 1985.

[28] S. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. C. Masiero, “Mutation analysis testing for finite state machines,” in *Proceedings of 1994 IEEE international symposium on software reliability engineering*, pp. 220–229, IEEE, 1994.

[29] P. E. Ammann, P. E. Black, and W. Majurski, “Using model checking to generate tests from specifications,” in *Proceedings second international conference on formal engineering methods (Cat. No. 98EX241)*, pp. 46–54, IEEE, 1998.

[30] E. Weyuker, “On testing non-testable programs,” *Computer Journal*, vol. 25, 11 1982.

[31] R. M. Hierons, “Testing from a Z specification,” *The Journal of Software Testing, Verification and Reliability*, vol. 7, no. 1, pp. 19–33, 1997.

[32] J. Dick and A. Faivre, “Automating the generation and sequencing of test cases from model-based specifications,” in *FME ’93, First International Symposium on Formal Methods in Europe*, (Odense, Denmark), pp. 268–284, Springer-Verlag, Lecture Notes in Computer Science 670, 19-23 April 1993.

[33] N. Li and J. Offutt, “Test oracle strategies for model-based testing,” *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 372–395, 2016.

[34] A. G. Clark, N. Walkinshaw, and R. M. Hierons, “Test case generation for agent-based models: A systematic literature review,” *Information and Software Technology*, vol. 135, p. 106567, 2021.

[35] P. McMinn, M. Stevenson, and M. Harman, “Reducing qualitative human oracle costs associated with automatically generated test data,” in *Proceedings of the First International Workshop on Software Test Output Validation*, pp. 1–4, 2010.

[36] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong, “Empirical studies of test-suite reduction,” *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.

[37] T. Y. Chen, S. C. Cheung, and S. M. Yiu, “Metamorphic testing: A new approach for generating next test cases,” 2020.

[38] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, “How effectively does metamorphic testing alleviate the oracle problem?,” *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 4–22, 2014.

[39] B. Brenner, M. A. Wainberg, and M. Roger, “Phylogenetic inferences on hiv-1 transmission: implications for the design of prevention and treatment interventions,” *AIDS (London, England)*, vol. 27, no. 7, p. 1045, 2013.

[40] Z. Q. Zhou and L. Sun, “Metamorphic testing of driverless cars,” *Communications of the ACM*, vol. 62, no. 3, pp. 61–67, 2019.

[41] A. F. Donaldson and A. Lascu, “Metamorphic testing for (graphics) compilers,” in *Proceedings of the 1st international workshop on metamorphic testing*, pp. 44–47, 2016.

[42] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, “Automated testing of graphics shader compilers,” *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[43] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, “A survey on metamorphic testing,” *IEEE Transactions on software engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[44] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou, “Metamorphic testing: A review of challenges and opportunities,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–27, 2018.

[45] T. Y. Chen, P.-L. Poon, and X. Xie, “Metric: Metamorphic relation identification based on the category-choice framework,” *Journal of Systems and Software*, vol. 116, pp. 177–190, 2016.

[46] A. Lascu, M. Windsor, A. F. Donaldson, T. Grosser, and J. Wickerson, "Dreaming up metamorphic relations: Experiences from three fuzzer tools," in *2021 IEEE/ACM 6th International Workshop on Metamorphic Testing (MET)*, pp. 61–68, IEEE, 2021.

[47] T. Chen, D. Huang, T. Tse, and Z. Q. Zhou, "Case studies on the selection of useful relations in metamorphic testing," in *In Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC'04).*, pp. 569–583, 11 2004.

[48] H. Liu, X. Liu, and T. Y. Chen, "A new method for constructing metamorphic relations," in *2012 12th International Conference on Quality Software*, pp. 59–68, 2012.

[49] U. Kanewala, J. M. Bieman, and A. Ben-Hur, "Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels," *Softw. Test. Verif. Reliab.*, vol. 26, pp. 245–269, nov 2015.

[50] F.-H. Su, J. Bell, C. Murphy, and G. Kaiser, "Dynamic inference of likely metamorphic properties to support differential testing," in *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*, pp. 55–59, IEEE, 2015.

[51] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei, "Search-based inference of polynomial metamorphic relations," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, (New York, NY, USA), p. 701–712, Association for Computing Machinery, 2014.

[52] B. Zhang, H. Zhang, J. Chen, D. Hao, and P. Moscato, "Automatic discovery and cleansing of numerical metamorphic relations," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 235–245, IEEE, 2019.

[53] R. Guderlei and J. Mayer, "Statistical metamorphic testing testing programs with random output by means of statistical hypothesis tests and metamorphic testing," in *Seventh International Conference on Quality Software (QSIC 2007)*, pp. 404–409, IEEE, 2007.

[54] M. S. Sadi, F.-C. Kuo, J. W. Ho, M. A. Charleston, and T. Y. Chen, "Verification of phylogenetic inference programs using metamorphic testing," *Journal of Bioinformatics and Computational Biology*, vol. 9, no. 06, pp. 729–747, 2011.

[55] L. Keele, "The statistics of causal inference: A view from political methodology," *Political Analysis*, pp. 313–335, 2015.

[56] D. B. Rubin, "Causal inference using potential outcomes: Design, modeling, decisions," *Journal of the American Statistical Association*, vol. 100, no. 469, pp. 322–331, 2005.

[57] J. Pearl, "Causal diagrams for empirical research," *Biometrika*, vol. 82, pp. 669–688, 12 1995.

[58] M. A. Hernán and J. M. Robins, *Causal Inference: what if.* Boca Raton: Chapman & Hall/CRC, 2020.

[59] J. Pearl, *Causality*. Cambridge university press, 2009.

[60] P. W. Holland, "Causal inference, path analysis and recursive structural equations models," *ETS Research Report Series*, vol. 1988, no. 1, pp. i–50, 1988.

[61] J. Pearl and D. Mackenzie, *The Book of Why*. Allen Lane, 2018.

[62] P. W. Holland, “Statistics and causal inference,” *Journal of the American statistical Association*, vol. 81, no. 396, pp. 945–960, 1986.

[63] K. Jager, C. Zoccali, A. Macleod, and F. Dekker, “Confounding: what it is and how to deal with it,” *Kidney international*, vol. 73, no. 3, pp. 256–260, 2008.

[64] D. Galles and J. Pearl, “Testing identifiability of causal effects,” *arXiv preprint arXiv:1302.4948*, 2013.

[65] M. A. Hernán and J. M. Robins, “Estimating causal effects from epidemiological data,” *Journal of Epidemiology & Community Health*, vol. 60, no. 7, pp. 578–586, 2006.

[66] J. Kendall, “Designing a research project: randomised controlled trials and their principles,” *Emergency medicine journal: EMJ*, vol. 20, no. 2, p. 164, 2003.

[67] N. Cartwright, “What are randomised controlled trials good for?,” *Philosophical studies*, vol. 147, no. 1, p. 59, 2010.

[68] J. M. Oakes, “Effect identification in comparative effectiveness research,” *eGEMs*, vol. 1, no. 1, 2013.

[69] R. A. Fisher, “Design of experiments,” *British Medical Journal*, vol. 1, no. 3923, p. 554, 1936.

[70] A. Nichols, “Causal inference with observational data,” *The Stata Journal*, vol. 7, no. 4, pp. 507–541, 2007.

[71] P. R. Rosenbaum, P. Rosenbaum, and Briskman, *Design of observational studies*, vol. 10. Springer, 2010.

[72] I. Kawaski and K. Lochner, “Socioeconomic status,” *Cancer Causes & Control*, pp. S39–S42, 1997.

[73] A. Van Loon, R. A. Goldbohm, I. Kant, G. Swaen, A. M. Kremer, and P. A. van den Brandt, “Socioeconomic status and lung cancer incidence in men in the netherlands: is there a role for occupational exposure?,” *Journal of Epidemiology & Community Health*, vol. 51, no. 1, pp. 24–29, 1997.

[74] U. S. S. G. A. C. on Smoking, *Smoking and Health: Report of the Advisory Committee to the Surgeon General of the Public Health Service*. No. 1103, US Department of Health, Education, and Welfare, Public Health Service, 1964.

[75] J. Droste, J. J. Weyler, J. P. Van Meerbeeck, P. A. Vermeire, and M. P. van Sprundel, “Occupational risk factors of lung cancer: a hospital based case-control study.,” *Occupational and Environmental Medicine*, vol. 56, no. 5, pp. 322–327, 1999.

[76] M. L. Petersen, K. E. Porter, S. Gruber, Y. Wang, and M. J. Van Der Laan, “Diagnosing and responding to violations in the positivity assumption,” *Statistical methods in medical research*, vol. 21, no. 1, pp. 31–54, 2012.

[77] T. J. VanderWeele, “Concerning the consistency assumption in causal inference,” *Epidemiology*, vol. 20, no. 6, pp. 880–883, 2009.

[78] D. H. Rehkopf, M. M. Glymour, and T. L. Osypuk, "The consistency assumption for causal inference in social epidemiology: when a rose is not a rose," *Current epidemiology reports*, vol. 3, no. 1, pp. 63–71, 2016.

[79] 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 (TKDD)*, vol. 15, no. 5, pp. 1–46, 2021.

[80] C. Wolf and H. Best, "The sage handbook of regression analysis and causal inference," *The SAGE Handbook of Regression Analysis and Causal Inference*, pp. 1–424, 2013.

[81] J. M. Wooldridge, *Introductory econometrics: A modern approach*. Cengage learning, 2015.

[82] L. C. Marsh and D. R. Cormier, *Spline regression models*. No. 137, Sage, 2001.

[83] S. McKinley and M. Levine, "Cubic spline interpolation," *College of the Redwoods*, vol. 45, no. 1, pp. 1049–1060, 1998.

[84] M. Alves, "Causal inference for the brave and true," 2022.

[85] S. Athey and S. Wager, "Estimating treatment effects with causal forests: An application," *Observational Studies*, vol. 5, no. 2, pp. 37–51, 2019.

[86] G. P. Zhang, "Neural networks for classification: a survey," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 30, no. 4, pp. 451–462, 2000.

[87] D. Westreich, J. Lessler, and M. J. Funk, "Propensity score estimation: machine learning and classification methods as alternatives to logistic regression," *Journal of clinical epidemiology*, vol. 63, no. 8, p. 826, 2010.

[88] V. Chernozhukov, D. Chetverikov, M. Demirer, E. Duflo, C. Hansen, W. Newey, and J. Robins, "Double/debiased machine learning for treatment and structural parameters," 2018.

[89] V. Chernozhukov, D. Chetverikov, M. Demirer, E. Duflo, C. Hansen, and W. Newey, "Double/debiased/neyman machine learning of treatment effects," *American Economic Review*, vol. 107, no. 5, pp. 261–265, 2017.

[90] J. M. Robins and Y. Ritov, "Toward a curse of dimensionality appropriate (coda) asymptotic theory for semi-parametric models," *Statistics in medicine*, vol. 16, no. 3, pp. 285–319, 1997.

[91] I. Díaz, "Machine learning in the estimation of causal effects: targeted minimum loss-based estimation and double/debiased machine learning," *Biostatistics*, vol. 21, no. 2, pp. 353–358, 2020.

[92] J. Siebert, "Applications of statistical causal inference in software engineering," *Information and Software Technology*, p. 107198, 2023.

[93] S. Lee, D. Binkley, R. Feldt, N. Gold, and S. Yoo, "Causal program dependence analysis," *arXiv preprint arXiv:2104.09107*, 2021.

[94] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[95] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, (New York, NY, USA), p. 73–84, Association for Computing Machinery, 2010.

[96] G. K. Baah, A. Podgurski, and M. J. Harrold, "Mitigating the confounding effects of program dependences for effective fault localization," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), p. 146–156, Association for Computing Machinery, 2011.

[97] G. Shu, B. Sun, A. Podgurski, and F. Cao, "Mfl: Method-level fault localization with causal inference," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 124–133, IEEE, IEEE, 2013.

[98] Z. Bai, S.-F. Sun, and A. Podgurski, "The importance of being positive in causal statistical fault localization: important properties of baah et al.'s csfl regression model," in *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)*, pp. 7–13, IEEE, 2015.

[99] G. W. Imbens, "The role of the propensity score in estimating dose-response functions," *Biometrika*, vol. 87, no. 3, pp. 706–710, 2000.

[100] A. Podgurski and Y. Küçük, "Counterfault: Value-based fault localization by modeling and predicting counterfactual outcomes," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 382–393, IEEE, 2020.

[101] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[102] Y. Küçük, T. A. Henderson, and A. Podgurski, "Improving fault localization by integrating value and predicate based causal inference techniques," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 649–660, IEEE, 2021.

[103] B. Johnson, Y. Brun, and A. Meliou, "Causal testing: understanding defects' root causes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 87–99, 2020.

[104] C. Dubslaff, K. Weis, C. Baier, and S. Apel, "Causality in configurable software systems," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 325–337, 2022.

[105] C. Baier, C. Dubslaff, F. Funke, S. Jantsch, R. Majumdar, J. Piribauer, and R. Ziemek, "From verification to causality-based explications," *arXiv preprint arXiv:2105.09533*, 2021.

[106] W. M. Johnston, J. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM computing surveys (CSUR)*, vol. 36, no. 1, pp. 1–34, 2004.

[107] P. Barham, A. Chowdhery, J. Dean, S. Ghemawat, S. Hand, D. Hurt, M. Isard, H. Lim, R. Pang, S. Roy, *et al.*, "Pathways: Asynchronous distributed dataflow for ml," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 430–449, 2022.

[108] A. Paleyes, S. Guo, B. Scholkopf, and N. D. Lawrence, "Dataflow graphs as complete causal graphs," in *2023 IEEE/ACM 2nd International Conference on AI Engineering–Software Engineering for AI (CAIN)*, pp. 7–12, IEEE, 2023.

[109] L. Merrick and A. Taly, "The explanation game: Explaining machine learning models using shapley values," in *Machine Learning and Knowledge Extraction: 4th IFIP TC 5, TC 12, WG 8.4, WG 8.9, WG 12.9 International Cross-Domain Conference, CD-MAKE 2020, Dublin, Ireland, August 25–28, 2020, Proceedings 4*, pp. 17–38, Springer, 2020.

[110] A. Paleyes and N. D. Lawrence, "Causal fault localisation in dataflow systems," in *Proceedings of the 3rd Workshop on Machine Learning and Systems*, pp. 140–147, 2023.

[111] H.-M. Heyn and E. Knauss, "Structural causal models as boundary objects in ai system development," in *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*, pp. 43–45, 2022.

[112] J. Pearl, "Causal inference in statistics: An overview," *Statistics Surveys*, vol. 3, pp. 96–146, 2009.

[113] Y. Liu, D. I. Mattos, J. Bosch, H. H. Olsson, and J. Lantz, "Bayesian causal inference in automotive software engineering and online evaluation," *arXiv preprint arXiv:2207.00222*, 2022.

[114] Z. Bai, G. Shu, and A. Podgurski, "Numfl: Localizing faults in numerical software using a value-based causal model," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–10, IEEE, 2015.

[115] M. Lechner *et al.*, "The estimation of causal effects by difference-in-difference methods," *Foundations and Trends® in Econometrics*, vol. 4, no. 3, pp. 165–224, 2011.

[116] G. W. Imbens and T. Lemieux, "Regression discontinuity designs: A guide to practice," *Journal of econometrics*, vol. 142, no. 2, pp. 615–635, 2008.

[117] 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)*, pp. 19–24, IEEE, 2021.

[118] H. Fang, H. Lamba, J. Herbsleb, and B. Vasilescu, ""this is damn slick!" estimating the impact of tweets on open source project popularity and new contributors," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 2116–2129, 2022.

[119] C. A. Furia, R. Torkar, and R. Feldt, "Towards causal analysis of empirical software engineering data: The impact of programming languages on coding competitions," *arXiv preprint arXiv:2301.07524*, 2023.

[120] E. Hunter, B. Mac Namee, and J. D. Kelleher, "A comparison of agent-based models and equation based models for infectious disease epidemiology.," in *AICS*, pp. 33–44, 2018.

[121] Ö. Özmen, J. J. Nutaro, L. L. Pullum, and A. Ramanathan, "Analyzing the impact of modeling choices and assumptions in compartmental epidemiological models," *Simulation*, vol. 92, no. 5, pp. 459–472, 2016.

[122] S. Daun, J. Rubin, Y. Vodovotz, and G. Clermont, "Equation-based models of dynamic biological systems," *Journal of critical care*, vol. 23, no. 4, pp. 585–594, 2008.

[123] W. Walter, *Ordinary Differential Equations*, vol. 182. Springer Science & Business Media, 2013.

[124] N. P. Johnson and J. Mueller, "Updating the accounts: global mortality of the 1918-1920" spanish" influenza pandemic," *Bulletin of the History of Medicine*, pp. 105–115, 2002.

[125] J. K. Taubenberger and D. M. Morens, "1918 influenza: the mother of all pandemics," *Revista Biomedica*, vol. 17, no. 1, pp. 69–79, 2006.

[126] R. Sanders and D. Kelly, "Dealing with risk in scientific software development," *IEEE software*, vol. 25, no. 4, pp. 21–28, 2008.

[127] C. C. Kerr, R. M. Stuart, D. Mistry, R. G. Abeysuriya, K. Rosenfeld, G. R. Hart, R. C. Núñez, J. A. Cohen, P. Selvaraj, B. Hagedorn, *et al.*, "Covasim: an agent-based model of COVID-19 dynamics and interventions," *PLOS Computational Biology*, vol. 17, no. 7, p. e1009149, 2021.

[128] I. for Disease Modelling, "Covasim." https://github.com/InstituteforDiseaseModeling/covasim, 2022.

[129] C. C. Kerr, D. Mistry, R. M. Stuart, K. Rosenfeld, G. R. Hart, R. C. Núñez, J. A. Cohen, P. Selvaraj, R. G. Abeysuriya, M. Jastrzębski, *et al.*, "Controlling COVID-19 via test-trace-quarantine," *Nature Communications*, vol. 12, no. 1, pp. 1–12, 2021.

[130] J. A. Cohen, D. Mistry, C. C. Kerr, and D. J. Klein, "Schools are not islands: Balancing COVID-19 risk and educational benefits using structural and temporal countermeasures," *medRxiv*, 2020.

[131] J. Panovska-Griffiths, C. C. Kerr, R. M. Stuart, D. Mistry, D. J. Klein, R. M. Viner, and C. Bonell, "Determining the optimal strategy for reopening schools, the impact of test and trace interventions, and the risk of occurrence of a second COVID-19 epidemic wave in the uk: a modelling study," *The Lancet Child & Adolescent Health*, vol. 4, no. 11, pp. 817–827, 2020.

[132] N. Scott, A. Palmer, D. Delport, R. Abeysuriya, R. Stuart, C. C. Kerr, D. Mistry, D. J. Klein, R. Sacks-Davis, K. Heath, *et al.*, "Modelling the impact of reducing control measures on the COVID-19 pandemic in a low transmission setting," *Med J Aust*, vol. 214, no. 2, pp. 79–83, 2020.

[133] T. C. Lucas, T. M. Pollington, E. L. Davis, and T. D. Hollingsworth, "Responsible modelling: unit testing for infectious disease epidemiology," *Epidemics*, vol. 33, p. 100425, 2020.

[134] S. T. Liang, L. T. Liang, and J. M. Rosen, "Covid-19: A comparison to the 1918 influenza and how we can defeat it," 2021.

[135] S. Segura, D. Towey, Z. Q. Zhou, and T. Y. Chen, "Metamorphic testing: Testing the untestable," *IEEE Software*, vol. 37, no. 3, pp. 46–53, 2018.

[136] M. Hernan, "Causal diagrams: Draw your assumptions before your conclusions," 2018.

[137] J. Pearl, "Direct and indirect effects," in *Probabilistic and causal inference: the works of Judea Pearl*, pp. 373–392, 2022.

[138] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. A. Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *Proceedings of SANER*, 2018.

[139] 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)*, pp. 153–164, IEEE, 2023.

[140] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software testing, verification and reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[141] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, 2004.

[142] R. S. Freedman, "Testability of software components," *IEEE transactions on Software Engineering*, vol. 17, no. 6, pp. 553–564, 1991.

[143] J. Tian and J. Pearl, "On the testable implications of causal models with hidden variables," *arXiv preprint arXiv:1301.0608*, 2012.

[144] 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 Trans. Software Eng. Method.*, Sept. 2022.

[145] M. L. Maciejewski and M. A. Brookhart, "Using instrumental variables to address bias from unobserved confounders," *Jama*, vol. 321, no. 21, pp. 2124–2125, 2019.

[146] P. Erdős, A. Rényi, *et al.*, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[147] "Cosmic ray: Variable inserter (fork)." https://github.com/AndrewC19/cosmic-ray/blob/master/src/cosmic_ray/operators/variable_inserter.py.

[148] "Cosmic ray: Variable remover (fork)." https://github.com/AndrewC19/cosmic-ray/blob/master/src/cosmic_ray/operators/variable_replacer.py.

[149] I. Tsamardinos, L. E. Brown, and C. F. Aliferis, "The max-min hill-climbing bayesian network structure learning algorithm," *Machine learning*, vol. 65, no. 1, pp. 31–78, 2006.

[150] M. de Jongh and M. J. Druzdzel, "A comparison of structural distance measures for causal bayesian network models," *Recent Advances in Intelligent Information Systems, Challenging Problems of Science, Computer Science series*, pp. 443–456, 2009.

[151] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[152] P. W. Tennant, E. J. Murray, K. F. Arnold, L. Berrie, M. P. Fox, S. C. Gadd, W. J. Harrison, C. Keeble, L. R. Ranker, J. Textor, *et al.*, "Use of directed acyclic graphs (DAGs) to identify confounders in applied health research: review and recommendations," *International Journal of Epidemiology*, vol. 50, no. 2, pp. 620–632, 2021.

[153] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[154] D. Malinsky and D. Danks, "Causal discovery algorithms: A practical guide," *Philosophy Compass*, vol. 13, no. 1, p. e12470, 2018.

[155] Q. Tao, W. Wu, C. Zhao, and W. Shen, "An automatic testing approach for compiler based on metamorphic testing technique," in *2010 Asia Pacific Software Engineering Conference*, pp. 270–279, IEEE, 2010.

[156] S. Chiba and M. Nishizawa, "An easy-to-use toolkit for efficient java bytecode translators," in *International Conference on Generative Programming and Component Engineering*, pp. 364–376, Springer, 2003.

[157] M. A. Hernán, S. Hernández-Díaz, M. M. Werler, and A. A. Mitchell, "Causal knowledge as a prerequisite for confounding evaluation: an application to birth defects epidemiology," *American journal of epidemiology*, vol. 155, no. 2, pp. 176–184, 2002.

[158] N. Vousden, R. Ramakrishnan, K. Bunch, E. Morris, N. Simpson, C. Gale, P. O'Brien, M. Quigley, P. Brocklehurst, J. J. Kurinczuk, *et al.*, "Impact of sars-cov-2 variant on the severity of maternal infection and perinatal outcomes: Data from the uk obstetric surveillance system national cohort," *MedRxiv*, 2021.

[159] C. Glymour, K. Zhang, and P. Spirtes, "Review of causal discovery methods based on graphical models," *Frontiers in genetics*, vol. 10, p. 524, 2019.

[160] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz— open source graph drawing tools," in *Graph Drawing* (P. Mutzel, M. Jünger, and S. Leipert, eds.), (Berlin, Heidelberg), pp. 483–484, Springer Berlin Heidelberg, 2002.

[161] J. Pearl and T. S. Verma, "A theory of inferred causation," in *Studies in Logic and the Foundations of Mathematics*, vol. 134, pp. 789–811, Elsevier, 1995.

[162] C. Cinelli and C. Hazlett, "Making sense of sensitivity: Extending omitted variable bias," *Journal of the Royal Statistical Society Series B-Statistical Methodology*, vol. 82, no. 1, pp. 39–67, 2020.

[163] T. J. VanderWeele and P. Ding, "Sensitivity analysis in observational research: introducing the e-value," *Annals of internal medicine*, vol. 167, no. 4, pp. 268–274, 2017.

[164] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. R. Ramakrishnan and J. Rehof, eds.), (Berlin, Heidelberg), pp. 337–340, Springer Berlin Heidelberg, 2008.

[165] J. Textor and M. Liskiewicz, "Adjustment criteria in causal diagrams: An algorithmic perspective," *arXiv preprint arXiv:1202.3764*, 2012.

[166] S. Wager and S. Athey, "Estimation and inference of heterogeneous treatment effects using random forests," *Journal of the American Statistical Association*, vol. 113, no. 523, pp. 1228–1242, 2018.

[167] P. Ralph, "Acm sigsoft empirical standards released," *SIGSOFT Softw. Eng. Notes*, vol. 46, p. 19, feb 2021.

[168] E. Bareinboim and J. Pearl, "Causal inference and the data-fusion problem," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 113, no. 27, pp. 7345–7352, 2016.

[169] C.-H. Luo and Y. Rudy, "A model of the ventricular cardiac action potential. Depolarization, repolarization, and their interaction.," *Circulation Research*, vol. 68, no. 6, pp. 1501–1526, 1991.

[170] J. P. Kleijnen, "Verification and validation of simulation models," *European journal of operational research*, vol. 82, no. 1, pp. 145–162, 1995.

[171] F. Sarrazin, F. Pianosi, and T. Wagener, "Global sensitivity analysis of environmental models: Convergence and validation," *Environmental Modelling & Software*, vol. 79, pp. 135–152, 2016.

[172] E. T. Chang, M. Strong, and R. H. Clayton, "Bayesian sensitivity analysis of a cardiac cell model using a Gaussian process emulator," *PloS one*, vol. 10, no. 6, p. e0130252, 2015.

[173] C. E. Rasmussen, "Gaussian processes in machine learning," in *Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2 - 14, 2003, Tübingen,*

Germany, August 4 - 16, 2003, Revised Lectures (O. Bousquet, U. von Luxburg, and G. Rätsch, eds.), (Berlin, Heidelberg), pp. 63–71, Springer Berlin Heidelberg, 2004.

[174] M. H. Grider, R. Jessu, and R. Kabir, "Physiology, action potential," 2019.

[175] M. Stein, "Large sample properties of simulations using Latin hypercube sampling," *Technometrics*, vol. 29, no. 2, pp. 143–151, 1987.

[176] cellML, "cellml: Luo-rudy 1991." https://models.cellml.org/exposure/456b07d6a7a5b45ed71caad0ea2c0b9d, 2022.

[177] J. W. Nevin, F. Vaquero-Caballero, D. J. Ives, and S. J. Savory, "Physics-informed gaussian process regression for optical fiber communication systems," *Journal of Lightwave Technology*, vol. 39, no. 21, pp. 6833–6844, 2021.

[178] C. E. Rasmussen, C. K. Williams, *et al.*, *Gaussian processes for machine learning*, vol. 1. Springer, 2006.

[179] J. H. Stock, M. W. Watson, *et al.*, *Introduction to econometrics*, vol. 104. Addison Wesley Boston, 2003.

[180] K. Benoit, "Linear regression models with logarithmic transformations," *London School of Economics, London*, vol. 22, no. 1, pp. 23–36, 2011.

[181] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE transactions on software engineering*, no. 3, pp. 178–187, 1978.

[182] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[183] A. Holzinger, G. Langs, H. Denk, K. Zatloukal, and H. Müller, "Causability and explainability of artificial intelligence in medicine," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 9, no. 4, p. e1312, 2019.

[184] N. Burkart and M. F. Huber, "A survey on the explainability of supervised machine learning," *Journal of Artificial Intelligence Research*, vol. 70, pp. 245–317, 2021.