
Proceedings Paper:
Clark, A.G., Foster, M., Walkinshaw, N. et al. (2023) Metamorphic testing with causal graphs. In: 2023 IEEE Conference on Software Testing, Verification and Validation (ICST). 2023 IEEE Conference on Software Testing, Verification and Validation (ICST), 16-20 Apr 2023, Dublin, Ireland. Institute of Electrical and Electronics Engineers (IEEE). ISBN: 9781665456678.

https://doi.org/10.1109/ICST57152.2023.00023

Metamorphic Testing with Causal Graphs

Andrew G. Clark, Michael Foster, Neil Walkinshaw, Robert M. Hierons
Department of Computer Science, University of Sheffield
{agclark2, m.foster, n.walkinshaw, r.hierons}@sheffield.ac.uk

***Abstract*—Metamorphic testing provides a means by which to generate succinct test oracles that can apply to large input spaces. For this it depends on the formulation of metamorphic relations, which generally require extensive domain expertise and human input. To address this problem, we present a model-based testing approach that can automatically generate metamorphic relations and associated tests. Our approach is motivated by the observation that metamorphic testing is a fundamentally causal task. We show how it is possible to leverage lightweight graph-based modelling techniques from the field of causal inference to specify causal properties of the system-under-test. Through a series of controlled experiments, we find that the proposed approach is robust to misspecification and can test evasive causal relationships (i.e. those that are difficult to exercise and observe) when combined with an appropriate test generation strategy. We also apply the approach to two case studies from the Defects4J framework with known bugs that affect causal behaviour. The results of these case studies suggest that the approach is not only useful for catching bugs affecting causal structure, but also alerting the user to inaccuracies in the specification.**

I. Introduction

Metamorphic testing provides an approach for reasoning about the correctness of a system's output in terms of changes to its inputs [1]. In this way, a single "metamorphic relationship" can be used to succinctly summarise the expected results for a potentially infinite range of inputs, as opposed to the conventional approach of writing test oracles that check the results for specific individual inputs. As a result, metamorphic testing has been successfully applied to various classes of systems that have traditionally been perceived as being particularly hard to test [2], such as machine learning systems [3] and compilers [4].

One barrier to the widespread application of metamorphic testing is the difficulty of formulating metamorphic relationships [2], [1]. Although several advances into semi-automatic metamorphic relation construction have been made [1], existing approaches have significant limitations. Techniques that reverse-engineer metamorphic relations from executions [5] risk reverse-engineering faulty relations if the underlying code is faulty. Other approaches are restricted to specific domains [6]. 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.

The approach proposed in this paper 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, there are several powerful approaches to modelling and reasoning about causality, which have become particularly widespread in areas such as epidemiology [7]. The authors are, however, not aware of any efforts to use these approaches to systematically generate metamorphic tests.

In this paper we propose a causal, model-based approach to the generation of metamorphic relations and test cases. Our approach is centred around causal directed acyclic graphs (DAGs) [8], a well-established and extremely simple graphical model that has been used extensively to model causal relationships in a wide range of disciplines. We show how causal DAGs can be used to model the expected causal relationships between inputs and outputs in a software system, and how these can be used in turn to automatically generate metamorphic relations and their test cases. Through a series of controlled experiments and real-world case studies, we then evaluate the ability of our approach to detect a specific class of bugs that modify the causal structure of the program-under-test and that appear in all forms of software.

Overall, 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 structural causal bugs under increasingly challenging conditions.
- We apply our technique to real-world software from the Defects4J framework with known structural causal bugs.

The remainder of this paper is structured as follows. Section II introduces a motivating example and provides the necessary background for this paper. Section III defines causal bugs and provides evidence that such bugs are common in software engineering. Section IV describes our process for creating metamorphic relations from causal DAGs. Section V presents an empirical evaluation of our technique before discussing the results in Section VI and demonstrating its application to two real-world case studies in Section VII. Section VIII then discusses related works in the literature and Section IX concludes the paper.

II. Background

This section discusses the necessary background for this work in the context of a toy example. Consider a simple program which calculates a person's BMI from their weight and height, and outputs a classification of "overweight", "underweight" or "healthy weight" based on this. Our goal is to test this program.

Metamorphic Testing: Determining whether a program produces the correct output in response to a given input depends on the availability of a "test oracle". In many practical situations, however, the intended behaviour of the software might be poorly documented or understood. The resulting challenge of providing a reliable verdict on a test execution is known as the "oracle problem" [9].

Metamorphic testing [10] is a promising solution to this problem [11]. The overarching concept is to consider an input and a transformation to be applied to that input. The expected output behaviour is then characterised by a corresponding transformation in the output space. This allows us to define the oracle in relative terms, which is often simpler than determining if a given output is correct. Collectively, these components are specified in what is known as a "metamorphic relation" (MR). We provide a formal definition below.

Definition 1. A *metamorphic relation* is a functional property of the system-under-test, S, that expresses how a given *source test case* (x_s) can be transformed into a *follow-up test case* (x_f) in such a way that some property over the outputs of these test cases can be anticipated, such as equality: $S(x_s) = S(x_f)$.

Let us consider an MR for our BMI example. For a person of weight w and height h, $BMI = w/h^2$. From this formula, we can derive the mutation relation $BMI(kw, h) = kBMI(w, h)$ using basic arithmetic. This gives us our source test case as $\{W = w, H = h\}$ and we can generate a follow-up test case $\{W = kw, H = h\}$. We then simply run both the source and follow-up test cases for various concrete values of w, h, and k and check if their outputs are equivalent.

Metamorphic testing is appealing because it presents an intuitive means by which to succinctly express properties against which to check software correctness [1]. Using MRs, the generation of test sets is often straightforward, especially with the help of tools such as JFuzz [12] and approaches such as iterative metamorphic testing [13].

A key barrier to the application of metamorphic testing lies in the construction of MRs [1]. Although techniques have been developed to aid MR construction [1], [2], they tend to rely heavily on the tester's intuition or on the availability of a reasonably expansive and established specification [14].

Causal Graphs: Our work is based on the observation that metamorphic testing is an inherently *causal* activity. Drawing on this similarity, we leverage well-established causal modelling approaches to develop a systematic approach for constructing and testing MRs. Here we provide the relevant background on graphical causal models.

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, such as large-scale software systems.

To address this problem, researchers in the field of causal inference developed a lightweight graphical model known as the causal graph [7], [8]. Causal graphs 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. Examples can be seen in Figures 1b, 5, and 7.

Definition 2. A causal graph G is a directed acyclic graph (DAG) $G = (V, E)$ comprising a set of nodes representing random variables, V, and a set of edges, 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.

Note that, unlike frameworks such as structural equation modelling [15], causal graphs do not specify the functional form of causal relationships, merely their presence, absence, and direction. From a testing standpoint, the lightweight nature of this model is desirable as it requires less input from the user.

Causal and Independence Relations: Causal graphs explicitly indicate which variables may (and, conversely, may not) affect each other by the presence (or absence) of edges. If a change to some variable of interest X (referred to a *treatment*) has no bearing on another variable Y (referred to as an *outcome*), i.e. $P(X \mid Y) = P(X)$, the two are said to be *independent*. In a situation where $P(X \mid Y, Z) = P(X \mid Y)$, we say that X and Y are *conditionally independent* given Z. Conversely, if X can be changed in a way that evokes a response in Y, we say that X causes Y.

Interventions: Causality concerns the effect of changes, actions, and policies; such terms are commonly referred to as *interventions*. Informally, an intervention is an external change to some variable in a system that forces that variable to take a specific value. We denote the act of intervening on a variable, X, to take a value x using Pearl's *do* notation [8]: $do(X = x)$. Viewed on a causal graph, $do(X = x)$ removes all incoming edges to X as it overrides the causal information flowing through the causes of X. Furthermore, the value that an output Y takes upon performing $do(X = x)$ is denoted as $Y^{X=x}$.

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 [16].

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 [7]. For example, in the causal graph $X \rightarrow Z \rightarrow Y$, 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 x', we simply measure the outcome for both values of the treatment, X, and observe the resulting change in Y. That is, $TE = Y^{X=x'} - Y^{X=x}$.

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 block the flow of information along all mediating paths. This is achieved by "fixing" the value of at least one mediator, Z, on this path. That is, $DE = Y^{X=x',Z=z} - Y^{X=x,Z=z}$.

The intuition here is to block all paths between X and Y apart from the path $X \rightarrow Y$ itself [16]. We refer to the set of "fixed" variables as an *adjustment set*.

Metamorphic Testing and Causality: At this point, we can draw a parallel with our definition of an MR (Definition 1). Essentially, an MR expresses the expected effect of a transformation defined by the pair of interventions $do(X = x_s)$ and $do(X = x_f)$, where x_s and x_f are the source and follow-up test cases, respectively. When viewed in this way, we can re-frame metamorphic testing to the task of measuring the *causal effect* of changing X from x_s to x_f on some output(s) Y.

III. Causal Bugs

Software bugs are commonly characterised in terms of "cause and effect" [17]. A particular parameter configuration might *cause* an execution path that leads to an erroneous output value. A particular GUI option might be implemented in the back-end, and ticking it might not have the *effect* on the output that the user might expect.

Such faults appear to be common. To explore the extent to which this is the case, we focus on Java units as a specific type of SUT. As a basis for our preliminary analysis, we refer to the analysis carried out by Sobreira et al. [18], who classified the fixes for each bug in the Defects4J repository [19] (which contains details of 395 bugs from six open source Java projects). Examples of their classifications include "method call addition" or "wrapping with an `if` statement".

Out of the 81 classes of repair they identified, 12 imply a change to the underlying causal structure of the program between the buggy and fixed versions. The majority of these involve the addition or removal of program variables (method parameters, class instantiations, etc.), corresponding to the addition or removal of nodes in the underlying causal structure. Additionally, the "wrong variable reference" repair suggests a change in the relationships between existing variables in the program, corresponding to a change in the edges of the underlying causal structure. Of the 395 bugs in Defects4J, 228 of them were tagged with at least one structural repair. We shall refer to such bugs as *structural causal bugs*.

Even when the causal structure of a program is as expected, there may still a problem with the precise functional form of the relationships between inputs and outputs. Changing an input may cause changes in the anticipated outputs, but the nature of that change (the value of the output with respect to the input) may be incorrect. We refer to such bugs as *functional causal bugs*. These require a more fine-grained test oracle than the one considered in this paper, and are left to future work. In Defects4J, we identified three repair categories as being related to functional causal bugs: arithmetic expression modification, conditional expression modification, and changes to constants. These suggest that the causal structure remains the same between the buggy and fixed program versions, but that the functional form of the relationships changes. 80 of the bugs in Defects4J are tagged with at least one of these categories, 51 of which are also tagged with a structural repair.

IV. Causal Metamorphic Testing

This paper is based on the observation that there is a strong relationship between causal DAGs and the type of behaviours that are expressed in MRs. Both are fundamentally based on the notion of a "causal effect". Implicitly, MRs (Definition 1) incorporate a pair of interventions that transforms a given input (from the source to the follow-up) 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) similarly incorporates the fact that a causal relationship between two variables occurs if an intervention on one may *cause* the other to change.

In this paper we explore the relationship between the two areas of metamorphic testing and (graph-based) causal reasoning. This raises the prospect of taking advantage of the significant amount of research on graph-based causal reasoning to identify and test potential MRs.

A. Causal DAGs as a Software Model

To apply causal DAGs to metamorphic testing, we need to model the Software Under Test (SUT) in the form of a causal DAG. For this we consider a simple abstraction:

- 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 SUT.

Definition 3. A *Causal Software Graph (CSG)* is an extension of a causal DAG (see Definition 2) defined as a tuple (V, E, I, O). Here, V and E correspond to the nodes and edges of the causal graph, 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[1] since inputs must be provided before any computation can be done.

Where software inputs are known to be independent, we may also assume $\forall (x, y) \in E.\, \neg (x \in I \wedge y \in i)$. 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 SUT. From this we can determine the set I: the input parameters and environmental 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 SUT. We discuss this in Section IV-B.

The edges between the variables represent our expectation of how variables should affect each other. As with most model-based testing techniques [20], the resulting model captures

[1] 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 ← ∅;
3      for x ∈ V do
           /* For every pair of variables that could feasibly
              form a causal edge                                  */
4          for y ∈ (V \ (I ∪ {x})) do
5              adjustmentSet ← identify(x, y, E);
6              for i = 0; i < N; i = i + 1 do
                   // Create a source input for variables in I
7                  source ← generateRandom(I, adjustmentSet);
                   /* Create a follow-up input, where variable x
                      is transformed to a different value          */
8                  followUp ← transform(source, x, adjustmentSet);
9                  if (x, y) ∈ E then
                       /* If (x, y) is a causal edge, changing
                          X should cause Y to change              */
10                     MT ← MT ∪ {(source, followUp, y, ≠)};
11                 else
                       /* If (x, y) is not a causal edge,
                          changing X should not affect Y          */
12                     MT ← MT ∪ {(source, followUp, y, =)};
13     return MT;
```

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

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 SUT, 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 V.

B. Generating Metamorphic Test Cases from a CSG

Algorithm 1 shows how a CSG can be used to generate 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.

The approach iterates through every pair of nodes (x, y), where y is not an input (lines 3 and 4). We here assume $\forall(x, y) \in E.\neg(x \in I \wedge y \in I)$ from Definition 3, although we could simply iterate over all node pairs in $V \times V$. For each pair of nodes, we proceed to identify an adjustment set (line 5) for the desired effect measure (see Section II), enabling the causal effect to be isolated. We then generate the source test case using the *generateRandom* function (line 7), which samples a random value for each variable in I and the adjustment set. The follow-up test case is obtained by applying the *transform* function (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. Finally, we equip our metamorphic test case with an oracle. If $(x, y) \in E$, we check for inequality between the source and follow-up outputs (line 10) since the causal edge means we should be able to observe a change in the outcome y. Conversely, if $(x, y) \notin E$, we check for equality instead (line 12) as, if there is no causal effect, there should be no change in y.

To execute our test cases, we need to be able to intervene on all variables x in the SUT for which we have a metamorphic test (x, y). While this is trivial for variables in I, we may also need to access and modify the values of variables in O. We refer to such variables 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 SUT. If, for example, the SUT 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 testability transformations [21] may be required (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 [22]. 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 *infer* causal effects involving hidden variables from existing data [23], [24]. This requires the identification of a more advanced adjustment set that accounts for particular sources of bias in the data, such as confounding and selection bias [25]. One such approach is to apply a graphical criterion known as d-separation [8]. We do not apply these techniques in this work, but plan to extend our solution to accommodate this functionality in future work.

C. Correctness and Performance

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

- **A1** *No model misspecification*: If the CSG perfectly reflects the causal structure of the SUT, we can generate an MR capable of exercising every causal and independence relationship.
- **A2** *Injective causal relationships*: If every causal relationship $X \rightarrow Y$ in the CSG is implemented in the SUT as an injective function (i.e. $f(x_s) = f(x_f) \implies x_s = x_f$), then, for a given MR, 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 3), 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 MRs that cover all causal and independence relations in the SUT.

If **A2** holds, we can verify each MR 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 $X \rightarrow Y$ 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 SUT 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 (**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 MRs and tests.

Misspecification is not the only way **A1** can be undermined. As discussed in Section IV-B, the (lack of) testability of the SUT 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 SUT cannot be controlled or observed [22].

It is also worth commenting here on the cost of our technique. While our solution essentially reduces the problem of specifying MRs to drawing a causal DAG, this process incurs its own overhead; for complex and unwieldy systems, drawing a DAG may be a non-trivial and time-consuming task. The exact cost of constructing a DAG is hard to quantify. However, the increasing adoption of causal DAGs as a lightweight domain model [26] across a range of fields such as epidemiology [27], [28] 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.

V. Evaluation

In Section IV, 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 MRs and ability to detect structural causal bugs.

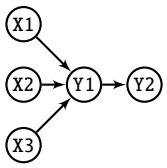
RQ2 *How robust to increasing amounts of non-injective causal relationships is causal metamorphic testing?*
When causal relationships are not injective, we face the additional challenge of finding a fault-revealing intervention, as the causal effect is only exercised by certain configurations. This research 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.

A. Experimental Setup

To answer **RQ1** and **RQ2**, we require software with a ground-truth CSG. To this end, we generate random CSGs and generate synthetic programs that reflect their causal structure, as in Figure 1. We then use mutation analysis to inject artificial bugs that either add or delete a causal relationship in the implementation. We now describe the setup for each of these steps in greater detail.

```
def program(X1,X2,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) CSG.

Fig. 1: A synthetic CSG and python code reflecting the causal structure.

CSG Generation: We first generate a random undirected graph using the Erdős-Rényi model [29], 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 1b. We then generate tests from the CSGs as described in Section IV-B, sampling parameter values from the range [-10, 10].

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 1b, 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.

For the purposes of answering **RQ2**, we add a "conditional behaviour" parameter, p_c to determine how difficult the causal effects are to observe. This 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].

For example, consider the if-then-else statement surrounding the definition of `Y1` in Figure 1a. 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 $X2$. Therefore, the source and follow-up test case must satisfy the predicate `X1 + X3 <= 2`, increasing the challenge of observing the causal effect of `X2` on `Y1`.

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 [30], [31].

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 $X \rightarrow Y$ in a causal graph (Definition 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`. This also applies to variables appearing in the predicates generated for **RQ2**. 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 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 DAG (i.e. without cycles, outputs causing inputs, or inputs causing inputs).

B. Methodology

Each experiment has three parameters n, p_e, and p_c, as discussed in Section V-A). 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.

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) [32], [33]. 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 then generate our metamorphic relations and tests using the test generation outlined in Section IV-B, before applying the mutation analysis approach described in Section V-A. We then plot the relationship between mutation score and SHD.

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 complexity [34] of the program and study the relationship between the mutation score and the McCabe 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 on the strata of data with $SHD = 0$, containing data from only the ground truth DAG (i.e. no misspecification).

VI. Results

Although we controlled the size n of the generated CSGs, there were no apparent differences between the corresponding results that would have a baring on the outcome of the RQs. Thus, we focus our discussion on the results for $n = 20$. Results for the other sizes can be found in our repository[2].

A. RQ1: Robustness to misspecification

The results in Figure 2 show that, as the CSG is increasingly misspecified (relaxing assumption **A1** from Section IV), the mutation score falls proportionally to 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

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

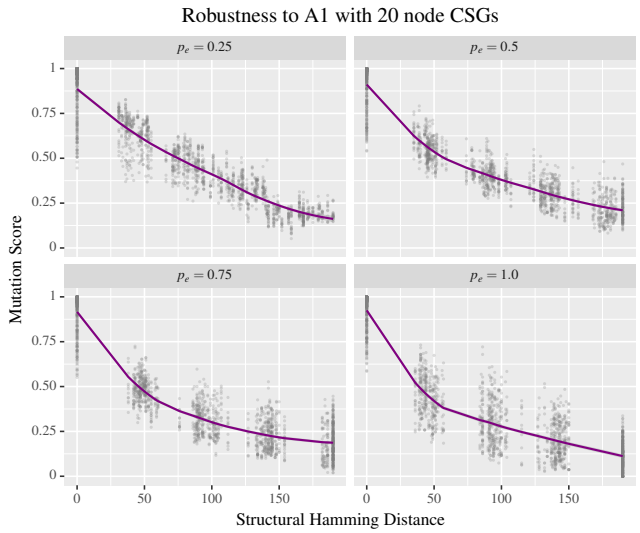


Fig. 2: Results for **RQ1** demonstrating the relationship between structural hamming distance (SHD) and mutation score for CSGs with 20 nodes and different edge densities (p_e).

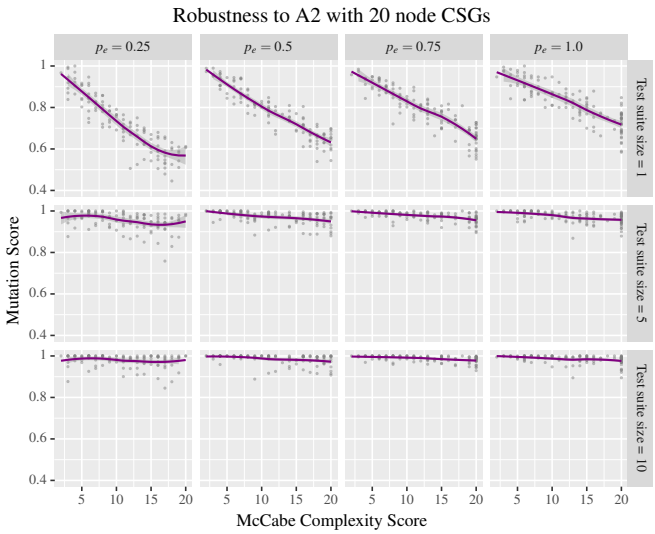


Fig. 3: Results for **RQ2** demonstrating the relationship between McCabe complexity and mutation score for CSGs with 20 nodes and different edge densities (p_e).

effects of the causal mutation operators on the adjustment sets of the generated MRs. When a spurious causal edge is added, the adjustment sets of MRs 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, MRs 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 [26].

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.

B. RQ2: Robustness to subtle causal relationships

Figure 3 shows that, as the McCabe complexity increases (relaxing assumption **A2** from Section IV) the mutation score falls when using just a single test case. For 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.

These findings indicate that faults residing in non-injective causal relationships (introduced here by conditional behaviour) can be notably harder to detect. However, they 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.

RQ2: Our results indicate that the fault detection capability of the proposed approach can be adversely affected by the presence of non-injective causal relationships. However, this issue may be mitigated by more rigorous test sets.

C. Threats to validity

External threats to validity: The main threat to external validity is that we generate synthetic programs in a parameterised, controlled manner. This was necessary to produce programs with a known causal structure, however they are unlikely to be representative of realistic programs. This concern motivated our case studies in Section VII where we apply our technique to two real programs from the Defects4J repository.

Another threat is that we employed a random test generation strategy to explore how the robustness to non-injective causal relationships varies with the size of a test suite. It is probable that we would have achieved better results by using a more advanced coverage-driven approach. 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 randomise 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 mitigate this threat by repeating each configuration 30 times as per [35].

Secondly, the metrics we select 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 [36], and McCabe complexity is a common metric for measuring software branching behaviour.

It is also worth acknowledging that we were not able to compare our approach to a baseline or existing techniques such as [4], [14]. Since there are very few MR generation techniques in the literature, there does not seem to be a commonly accepted baseline or set of comparison metrics. We leave this to future work.

VII. Case studies

In Section V, 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 changes as assumptions **A1** and **A2** from Section IV deteriorate. In this section, we apply our technique to detect structural causal bugs from two systems in the Defects4J repository [19]. The implementation of these case studies can be found in our repository[3].

A. 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[4], 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:

$$variance = \frac{\sum_{i=begin}^{begin+length} Weights_i(Values_i - mean^2)}{(\sum_{i=begin}^{begin+length} Weights_i) - 1} \quad (1)$$

The implementation of this method contains a structural causal bug, as shown in Figure 4. During the computation of the denominator in Equation (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];
      }
```

Fig. 4: A diff showing a structural causal bug located in the `evaluate` method of the `Variance` class.

[3]https://github.com/AndrewC19/DAG-MT-case-studies

[4]https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/stat/descriptive/moment/Variance.html

Causal Software Graph: The CSG shown in Figure 5 uses only information that is available in the 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 instrument the implementation to expose the corresponding values in the implementation as it executes (elaborated below).

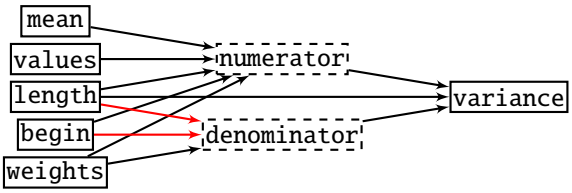


Fig. 5: CSG for `Variance.evaluate`, with the fault-causing relationships highlighted in red and hidden variables shown as dashed boxes.

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.

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 (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 address the latter of these concerns in the following case studies.

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: `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.

B. 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 thread-safe implementation for formatting and parsing dates that accounts for the user's locale[5].

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 6. 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();
  }
```

Fig. 6: 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 7 using only information available in the documentation. Nodes represent the variables used to instantiate the `FastDateFormat` object: `locale`, `timeZone`, and `rules`. 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 7 we constructed a total of 12 metamorphic relations, of which 7 are `ShouldCause` and 5 are `ShouldNotCause`. Using

[5]https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/time/FastDateFormat.html

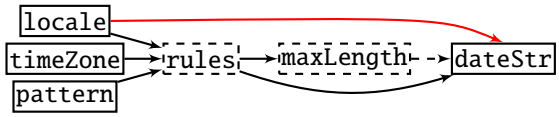


Fig. 7: A CSG for the `FastDateFormat.format` method, with the fault-causing relationship highlighted in red, a mis-specified edge shown by a dashed edge, and the hidden variables in dashed boxes.

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.

Instrumentation: Whereas the previous case study dealt with a single method, this implementation is spread across a class. The hidden variables in our CSG (`rules` and `maxLength`) ultimately turn out to be derivable from class variables as opposed to internal method variables, which makes the task of observing and manipulating them much easier. In practice, we used the documentation to develop an automated bytecode instrumentation tool for this.

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 7, `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 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.

VIII. Related Work

This section provides a review of work related to the core topics of this paper, namely the construction of MRs, graph-based test case generation, and causality in software engineering. We also briefly summarise methods designed to learn causal graphs from data.

Construction of Metamorphic Relations: Several papers in the literature present heuristics to aid users in creating MRs [37], [38], [39]. The main consensus is that source and

follow up test cases which are "further apart" by some metric are more likely to discover faults. Mayer and Guderlei [38] also find that MRs phrased as equalities are relatively weak, although Chen et al. [37] say that theoretically stronger MRs are not necessarily better at detecting real program defects.

More recent work [14] presents a method where users are given pairs of concrete parameter settings and asked to either generalise them into MRs or declare that no MR exists for the pair. Additionally, Liu et al. [40] suggest constructing new MRs by composing existing ones.

These methods all require significant input from the user. To address this, Zhang et al. [5] introduce a particle swarm optimisation based approach for automatically inferring polynomial MRs from program executions. This significantly reduces the burden on the user but has two key limitations. First, it requires a large amount of execution data from the SUT. Second, MRs inferred from execution data will reflect any bugs in the implementation, meaning we must now determine which of the inferred MRs capture the expected behaviour.

Test Case Generation from Directed Graphs: Vilkomir et al. [41], [42] present a method of generating combinatorial test cases using a DAG representing dependencies between program inputs. Edges are labelled with parameter values such that test paths through the graph represent test cases. Their technique differs from ours in several ways. Where their graphical models only illustrate dependencies between input parameters, ours may also include outputs and internal variables, allowing them to also serve as test oracles. Second, our models do not require the user to precisely describe parameter domains. Our approach also produces metamorphic test cases where Vilkomir et al.'s approach produces combinatorial ones.

Chen et al. [43] applied metamorphic testing to two bioinformatic programs that take a weighted directed graph representing gene-to-gene interactions as input. This work is fundamentally different from ours in that our DAGs model the expected behaviour of the SUT where Chen et al.'s are an input. Their work is also highly specialised to the field of gene regulatory behaviour, where our technique is domain agnostic.

Metamorphic Testing Tools: Several tools and techniques exist to assist in the practicalities of metamorphic testing. Zhu [12] developed JFuzz, a tool for unit testing Java programs via metamorphic testing. Given a class to test and a series of MRs (implemented as classes), this tool automates the generation of source and follow-up test cases.

Kanewala et al. [6] introduced a machine learning-based approach that *predicts* MRs based on a control flow graph and data dependency information. The approach has been shown to accurately predict whether a given function satisfies a library of pre-defined MRs commonly used for testing machine learning applications, but it does not aim to produce new ones.

Causality in Software Testing: There are a number of techniques that aim to establish causality in software systems. Johnson et al. [44] developed a tool to explain the root cause of faulty software behaviour by mutating existing tests to form a suite of minimally different tests that, contrary to the original, are not fault-causing. The passing and failing tests can then be compared to understand *why* a fault occurred.

Another relevant technique is cause-effect graphing [45], [46]. Here, input-output relationships are expressed in a variant of a combinatorial logic network called a cause-effect graph, created by manually extracting causes, effects, and boolean constraints from natural language specifications.

In a similar vein, dataflow testing [47] is a family of testing strategies designed to verify the interactions between each program variable's definition and its uses, referred to as def-use pairs. The aim here is to exercise def-use pairs with respect to some coverage criterion. Although not explicitly related to causality, the field shares a similar objective to our work, namely to verify *interactions* between variables.

Automatic Generation of DAGs: While manual creation of DAGs is widely accepted in fields such as epidemiology, methods exist that could (partially) automate this process. Causal discovery [36] aims to learn causal structures from data by exploiting asymmetries that separate association from causation [48]. Causal DAGs have also been generated via static analysis of source code [49], [50]. A fundamental weakness of both approaches is that inferred graphs represent the *actual system* rather than its *intended behaviour*, so reflect any bugs present in the implementation.

IX. Conclusions

While metamorphic testing presents a promising solution to the oracle problem, a key challenge is to devise MRs describing the software behaviour in terms of how *changes* to the input parameters bring about changes to the outputs. Existing techniques either rely heavily on the user [14], or infer relations using the program itself [5], meaning the inferred relations may reflect implementational bugs in the SUT.

In this paper, we proposed a model-based technique for generating metamorphic relations from a lightweight, intuitive graphical causal model of software behaviour. 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 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 revealed 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.

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. Furthermore, there is the potential to combine the proposed technique with more advanced test generation and selection techniques. A more comprehensive evaluation, application to further case studies, and perhaps a human usability study is also desirable to draw more general conclusions about our technique's performance.

References

[1] 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.

[2] 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.

[3] C. Murphy, G. Kaiser, L. Hu, and L. Wu, "Properties of machine learning applications for use in metamorphic testing," *Combining SOA and BPM Technologies for Cross-System Process Automation*, p. 867, 2008.

[4] 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*. IEEE, 2010, pp. 270–279.

[5] 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*, ser. ASE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 701–712. [Online]. Available: https://doi.org/10.1145/2642937.2642994

[6] 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, no. 3, pp. 245–269, nov 2015. [Online]. Available: https://doi.org/10.1002%2Fstvr.1594

[7] M. Hernan and J. Robins, *Causal Inference: What if*. CRC Press, 2020.

[8] J. Pearl, *Causality*. Cambridge university press, 2009.

[9] 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.

[10] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," 2020. [Online]. Available: https://arxiv.org/abs/2002.12543

[11] 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.

[12] H. Zhu, "Jfuzz: A tool for automated java unit testing based on data mutation and metamorphic testing methods," in *2015 Second International Conference on Trustworthy Systems and Their Applications*. IEEE, 2015, pp. 8–15.

[13] P. Wu, "Iterative metamorphic testing," in *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, vol. 1. IEEE, 2005, pp. 19–24.

[14] 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.

[15] R. B. Kline, *Principles and practice of structural equation modeling*. Guilford publications, 2015.

[16] H. Geffner, R. Dechter, and J. Y. Halpern, Eds., *Probabilistic and Causal Inference: The Works of Judea Pearl*, 1st ed. New York, NY, USA: Association for Computing Machinery, 2022, vol. 36.

[17] 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.

[18] 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.

[19] 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*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[20] 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.

[21] 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.

[22] R. S. Freedman, "Testability of software components," *IEEE transactions on Software Engineering*, vol. 17, no. 6, pp. 553–564, 1991.

[23] J. Tian and J. Pearl, "On the testable implications of causal models with hidden variables," *arXiv preprint arXiv:1301.0608*, 2012.

[24] A. G. Clark, M. Foster, B. Prifling, N. Walkinshaw, R. M. Hierons, V. Schmidt, and R. D. Turner, "Testing causality in scientific modelling software," 2022. [Online]. Available: https://arxiv.org/abs/2209.00357

[25] S. Greenland and J. Pearl, *Causal Diagrams*. John Wiley & Sons, Ltd, 2017, pp. 1–10. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118445112.stat03732.pub2

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] "Cosmic ray: Variable inserter." [Online]. Available: https://anonymous.4open.science/r/cosmic-ray-FORK258/src/cosmic_ray/operators/variable_inserter.py

[31] "Cosmic ray: Variable remover." [Online]. Available: https://anonymous.4open.science/r/cosmic-ray-FORK258/src/cosmic_ray/operators/variable_replacer.py

[32] 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.

[33] 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.

[34] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[35] 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.

[36] D. Malinsky and D. Danks, "Causal discovery algorithms: A practical guide," *Philosophy Compass*, vol. 13, no. 1, p. e12470, 2018.

[37] 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).*, 11 2004, pp. 569–583.

[38] J. Mayer and R. Guderlei, "An empirical study on the selection of good metamorphic relations," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, vol. 1, 2006, pp. 475–484.

[39] Y. Cao, Z. Q. Zhou, and T. Y. Chen, "On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions," in *2013 13th International Conference on Quality Software*, 2013, pp. 153–162.

[40] H. Liu, X. Liu, and T. Y. Chen, "A new method for constructing metamorphic relations," in *2012 12th International Conference on Quality Software*, 2012, pp. 59–68.

[41] S. A. Vilkomir, W. T. Swain, J. H. Poore, and K. T. Clarno, "Modeling input space for testing scientific computational software: A case study," in *Computational Science – ICCS 2008*, M. Bubak, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 291–300.

[42] S. A. Vilkomir, W. T. Swain, and J. H. Poore, "Combinatorial test case selection with markovian usage models," in *Fifth International Conference on Information Technology: New Generations (itng 2008)*, 2008, pp. 3–8.

[43] T. Y. Chen, J. W. Ho, H. Liu, and X. Xie, "An innovative approach for testing bioinformatics programs using metamorphic testing," *BMC Bioinformatics*, vol. 10, no. 1, jan 2009. [Online]. Available: https://doi.org/10.1186%2F1471-2105-10-24

[44] 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*, 2020, pp. 87–99.

[45] K. Nursimulu and R. L. Probert, "Cause-effect graphing analysis and validation of requirements," in *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative research*. Citeseer, 1995, p. 46.

[46] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The Art of Software Testing*. Wiley Online Library, 2004, vol. 2.

[47] T. Su, K. Wu, W. Miao, G. Pu, J. He, Y. Chen, and Z. Su, "A survey on data-flow testing," *ACM Comput. Surv.*, vol. 50, no. 1, mar 2017. [Online]. Available: https://doi.org/10.1145/3020266

[48] C. Glymour, K. Zhang, and P. Spirtes, "Review of causal discovery methods based on graphical models," *Frontiers in genetics*, vol. 10, p. 524, 2019.

[49] 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)*. IEEE, 2020, pp. 382–393.

[50] S. Lee, D. Binkley, R. Feldt, N. Gold, and S. Yoo, "Causal program dependence analysis," *arXiv preprint arXiv:2104.09107*, 2021.