This is a repository copy of *CMBMeTest:Generation of Test Suites Using Model-Based Testing Plus Constraint Programming and Metamorphic Testing*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/217614/

Version: Published Version

Article:

Castro-Cabrera, M. Carmen de, García-Dominguez, Antonio orcid.org/0000-0002-4744-9150 and Medina-Bulo, Inmaculada (2023) CMBMeTest:Generation of Test Suites Using Model-Based Testing Plus Constraint Programming and Metamorphic Testing. Electronics (Switzerland). 18. ISSN 2079-9292

https://doi.org/10.3390/electronics13010018

Reuse

This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here: https://creativecommons.org/licenses/

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Article

CMBMeTest: Generation of Test Suites Using Model-Based Testing Plus Constraint Programming and Metamorphic Testing

M. Carmen de Castro-Cabrera [1,*], Antonio García-Dominguez [2] and Inmaculada Medina-Bulo [1]

[1] Department of Computer Science, University of Cádiz, 11519 Cádiz, Spain; inmaculada.medina@uca.es
[2] Department of Computer Science, University of York, York YO10 5DD, UK; a.garcia-dominguez@york.ac.uk
* Correspondence: maricarmen.decastro@uca.es

Abstract: Various software testing techniques have been shown to be successful in producing high-quality test suites for software where the code is not accessible (black-box approach). Nevertheless, no method has been found to guide combining some of these in a general way. In this study, a test suite generation method for black-box software called CMBMeTest was created to respond to these challenges. It employs several coupled software testing techniques, namely, model-based testing (MBT), constraint programming (CP), and metamorphic testing (MT). CMBMeTest provides step-by-step instructions for using the information available (such as program specifications, inputs and outputs) to create an initial test suite that covers the model obtained, using a combination of MBT and CP (referred to as MBT+CP). Furthermore, using the metamorphic relations (MRs) of MT, a better test suite was produced from that initial test suite. The method allows particular stages to be iterated to improve the results by building new models and new MRs. A comprehensive case study was conducted, employing CMBMeTest to produce encouraging results. Mutation testing was used to evaluate the test suite, and the first round produced a high mutation score. A more detailed model was used to repeat the process, with similar outcomes.

Keywords: model-based testing; metamorphic relations; constraint solvers; constraint programming systems; test case generation; test suites

Citation: Castro-Cabrera, M.C.d.; García-Dominguez, A.; Medina-Bulo, I. CMBMeTest: Generation of Test Suites Using Model-Based Testing Plus Constraint Programming and Metamorphic Testing. *Electronics* **2024**, *13*, 18. https://doi.org/10.3390/electronics13010018

Academic Editors: Hsi-Min Chen and Shang-Pin Ma

Received: 15 November 2023
Revised: 10 December 2023
Accepted: 13 December 2023
Published: 19 December 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

One of the most important tasks in software testing is to generate an adequate set of test cases to detect the largest number of bugs in the program, i.e., to cover most of the implemented code. This task becomes a challenge when the source code of the software to be tested is inaccessible, i.e., when the testing approach uses a black box. Although several techniques can be used to test this type of software, independently or integrated with others, generic guides to applying them are scarce. This paper presents CMBMeTest, a method that addresses the challenge of generating coverage test suites using a black-box approach. For this purpose, a series of steps to be followed and several combined testing techniques are described. The method starts from the available program information, typically inputs, outputs and/or their specification, and derives a coverage test suite. In addition, tools are proposed to automate part of the process. A complete case study is developed by applying the proposed method, with the test suite generated with mutation testing being evaluated. The results obtained are promising, achieving a mutation score of more than 90%.

One of the most popular methods for testing software with the black-box approach is known as model-based testing (MBT), which can abstract away the specifics of the language or implementation to concisely describe their behavior [1]. Additionally, it is simpler to examine the program model's coverage criteria. This method is applied at numerous levels of software testing, particularly in system testing, and with a wide range of automation levels and tools. GraphWalker (GW) [2], which offers graphical model design and execution capabilities, is one example of an MBT tool. Moreover, having a set

of test cases that meets specific coverage requirements is helpful when testing software. Typically, some limitations on the program's expected inputs and outputs are established during the test case generation process in order to accomplish this coverage. One way to set input/output limitations is to use constraint programming (CP) to identify the desired outputs and solve the constraints required to pursue a particular path. The combination of MBT and CP techniques is called MBT+CP.

Furthermore, metamorphic testing (MT) has proven to be an effective technique in various fields and disciplines, including cloud computing [3], geographical information systems [4], chatbots [5], or distributed computations [6]. Metamorphic testing attempts to alleviate the test oracle problem (Section 2.4) and other problems (improving fault detection rates) through the use of metamorphic relations (MRs), which are known relationships between the inputs and outputs of multiple executions of the program under test [7]. As an example, if our software is computing $\sin(x)$, one MR could be that if $\sin(x) = y$, then $\sin(-x) = -y$: from a test case with input x and expected output y, we could derive one with input $-x$ and expected output $-y$.

This paper provides a full description of CMBMeTest: a guided method for generating high-coverage test suites with a black-box approach, from the program specification to the generated suite. Initially, a source test suite is generated through MBT+CP. Subsequently, MRs are designed and implemented (following the proof of concept from our prior proof-of-concept MR identification study [8]), and executed to obtain a better test suite (follow-up test suite). The suite generated is evaluated using auxiliary testing techniques.

We apply each step to a case study, MetaSearch [9], which is a Web Services Business Process Execution Language (WS-BPEL) composition. WS-BPEL has been chosen because there exist tools that can evaluate the quality of the tests produced by the approach via mutation testing, which seeds defects and checks whether the tests can detect them.

The contributions of this paper are as follows:

- A generic guided method to obtain a suite of quality test cases on programs where sources are unavailable, using MBT+CP and metamorphic testing.
- The test suite generation for a case study through MBT+CP combined techniques with the GraphWalker and MiniZinc tools.
- The design and implementation of metamorphic relations for a case study, employing CP in the MiniZinc constraint model language.
- New enhanced test suites for a case study, obtained through metamorphic relations.
- Validation of the quality of the generated test suite using the mutation testing technique.

This paper is structured as follows: Section 2 introduces background concepts in MBT, CP, and MT, and Section 3 explores related work in these areas. Section 4 describes the six steps of the proposed method in a language-agnostic manner. In Section 5, the method is illustrated and evaluated, step-by-step, by means of a case study. Section 6 discusses the results, while Section 7 analyzes the threats to validity. Finally, the last section presents the conclusions and future lines of research.

2. Background

This section describes certain concepts that are required to understand this work. We start with a general introduction to the definition and generation of test cases. We then briefly describe the testing techniques and tools involved in the process: model-based testing, GraphWalker, constraint programming, MiniZinc and metamorphic testing.

2.1. Test Case Definition and Generation

Beizer [10] considers that the goal of software testing depends on the maturity level of the software testing process in an organization. He lists five levels, of which this paper aims to achieve the fifth one: "Testing is a mental discipline that helps all IT professionals develop higher-quality software". To reach this level, as described by Ammann and Offutt [11] in one of the basic texts on software testing, the current approach follows two steps: building a model of the program under test (using structures and criteria such as input spaces, graphs,

logical expressions, and syntactic representations), and generating tests from this model by finding the various ways in which it can be exercised.

For any program, the possible number of inputs is typically immense, and it is not feasible to test them all at every stage of program development. Therefore, the test designer's goal will be to carefully sample these large input spaces, seeking the fewest tests that show the most problems. That is, there are two questions to be solved: how to search, and when to stop. To answer these questions, test requirements need to be considered. According to Ammann and Offutt [11] "a coverage criterion is a rule or collection of rules that imposes test requirements on a test set". They are specific elements of a software artifact that a test must satisfy. For example, coverage criteria require that the tests exercise the software in a certain number of ways.

The level of abstraction in testing can be raised by using mathematical structures. This is how model-driven test design (MDTD) emerged. MDTD decomposes the tests, according to the coverage criteria, into a sequence of small tasks (based on the previous structures) that simplify the generation of tests. These structures can be extracted from many software artifacts. For example, graphs can be extracted from UML use cases, finite state machines or source code, among others, and logical expressions can be extracted from decisions in source code, from conditionals in use cases.

Meanwhile, for test design, two complementary approaches should be considered (Chapter 2 of [11]): design based on code coverage criteria and design based on people (domain experts that can detect possible failures that the coverage criteria do not detect). While the former is more technical and mathematical (described above), the latter requires knowledge of the problem domain and experience in testing solutions for that domain.

Once these models are defined, it is relatively simple to automate the process of defining and generating test cases, achieving agility and reducing costs in software testing. Chapter 3 of Ammann and Offutt [11] defines test automation in a broad sense as: "The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions".

Taking into account this approach, many works have focused on how to generate high-quality test cases in an automated way. For example, Utting et al. [12] discuss the model-based testing process, defining a taxonomy that covers the key aspects of MBT proposals as well as tools to implement them. In addition, Fraser and Arcuri [13] present the results of an experiment on unit test generation using the EvoSuite system on several open-source and industrial projects.

2.2. Model-Based Testing

Model-based testing (MBT), which abstracts the details of a language or implementation to describe its behavior in a simple way, is one of the most frequently used methods to test software. By developing a model that captures the behavior of the system under test, model-based testing is an effective and flexible approach for testing software [1].

Additionally, it is simpler to verify the coverage requirements of the program model. This method is utilized with a wide range of automation levels and a variety of tools at various levels of software testing, particularly in system testing [14].

GraphWalker [2], which offers graphical model creation and execution capabilities, is an example of an MBT tool. GraphWalker has a web-based editor (Studio) to view and edit its graph-based models, and a command-line tool (CLI) to execute the model by traversing the graph.

GraphWalker graphs are finite state machines (FSMs) in which the states are represented by the vertices and the transitions by the edges. One vertex in the graph must be selected as the initial state, and a path generator expression must be provided: this expression determines how the graph is traversed and when to stop. For example, the `random(edge_coverage(100))` expression would tell GraphWalker to perform a random walk over the graph, and to stop when all edges have been covered. Likewise, the tool allows the inclu-

sion of guards and actions on transitions, which, respectively, limit when the transition can be triggered and execute code when the transition is triggered. Once the model is created and the different properties are defined, the model can be executed to check whether it behaves as expected. In GraphWalker Studio, the transitions and states are colored as you run through the diagram. Subsequently, and through the GraphWalker CLI tool, we obtain a set of paths that guarantee the proposed constraints and coverage.

2.3. Constraint Programming

In constraint programming (CP), a program comprises a set of constraints (a *model*) such that, when executed, it finds a solution that satisfies those constraints. The exact procedure is tasked to the *constraint solver*, a piece of software which implements the appropriate decision processes.

This paradigm emerged in the 1980s, although there was some prior work. In the 1990s, they were used more in practice, generally as an extension of logic programming languages: the approach was called constraint logical programming (CLP). They are, broadly speaking, software development environments and were mainly used to build applications for planning (personnel, production, etc.) and design problems [15]. The use of constraint programming systems (CPSs) is growing rapidly, and they are increasingly being implemented in a wider range of domains [16,17].

CPSs are based on different programming languages and paradigms, with there being many different tools. Kjellerstrand provided a comparison of the available CPS in [18]. For the present work, a number of different options were studied, considering the conciseness of the syntax, its ease of use, the community behind the tool, the quality of the documentation and their capabilities. Among others, JaCoP [19], Choco [20], Gecode [21], Tailor/Essence [22], and MiniZinc [23] were studied. Of these options, MiniZinc strikes a good balance of features: (i) many solvers can parse MiniZinc (including several versions of Gecode); (ii) it is concise, since it provides high-level elements and logical operators; (iii) it has ample documentation [24]; (iv) it has an active user community around it; and (v) it is available as open source.

MiniZinc is an open-source constraint modeling language [23] that can describe constraint resolution and optimization problems while allowing alternative solvers to be used. A graphical interface is included in the default distribution of MiniZinc for ease of use, as well as examples [25]. Input data is separated into its own data file, away from the model files. An *.mzn* model file and a *.dzn* data file are translated into a FlatZinc model, specific to the solver.

A MiniZinc model consists of a set of variables and parameter declarations, followed by a set of constraints and a `solve` statement (and, optionally, `output`). The parameters are set by the user in the data file, and the variables are calculated by the solver while solving the constraints. The solver can be told to simply find a solution that satisfies all constraints, or a solution that maximizes a particular objective. Constraints can be set on any variables in the model, while the language allows flexibility in terms of the operators to apply to those data. Listing 1 shows an example of a MiniZinc model with the solution (`mary = 9, jack = 3 and violet = 12`).

Listing 1. MiniZinc model example.

```
1  % Example problem about ages in MiniZinc.
2  %
3  % Mary is three times as old as Jack, the sum of their ages is that of Violet's,
4  % while the ages of the three of them add up to 24. How old are they?
5
6  int: n = 24; % constant
7  var 0..n: mary; % variable
8  var 0..n: jack;
9  var 0..n: violet;
```

```
10
11 constraint   mary = jack*3
12           /\
13       mary + jack = violet
14           /\
15       mary + jack + violet  = 24;  % constraint
16
17 solve  satisfy ;  % solve  statement
```

2.4. Metamorphic Testing

Metamorphic testing, as mentioned in the previous section, has been proposed to alleviate the oracle problem [26]. MT relies on the notion of a metamorphic relation (MR). In Chen et al. [7], an MR is defined as follows:

> Let f be a target function or algorithm. A metamorphic relation is a necessary property of f over a sequence of two or more inputs $\langle x_1, x_2, \ldots, x_n \rangle$, where $n \geq 2$, and their corresponding outputs $\langle f(x_1), f(x_2), \ldots, f(x_n) \rangle$. It can be expressed as a relation $R \subseteq X^n \times Y^n$, where $\subseteq$ denotes the subset relation, and X^n and Y^n are the Cartesian products of n input and n output spaces, respectively. Following standard informal practice, we may simply write $R(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n))$ to indicate that $\langle x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n) \rangle \in R$.

When the implementation is correct, program inputs and outputs are expected to satisfy necessary properties that are relevant to the underlying algorithms. Through MRs, it is possible to generate new test cases from the initial ones. As an example, take a function f, which, given m natural numbers, $n_0, n_1, \ldots, n_m$, calculates their mean. If the inputs are rearranged (for example, $n_0, n_2, n_1, \ldots, n_m$), the result must be the same, since it is a known property of the arithmetic mean. Formally, we can express this as

$$\begin{aligned} \mathrm{MR}_1 \quad \equiv \quad & (\forall x \, L_2 = \mathrm{perm}((n_0, n_1, n_2, \ldots n_m), x) \\ & \implies \mathrm{mean}(L_2) = \mathrm{mean}((n_0, n_1, n_2, \ldots n_m))) \end{aligned}$$

where $(n_0, n_1, n_2, \ldots n_m)$, is the original input and L_2 will be the follow-up test case input (the x-th permutation of $(n_0, n_1, n_2, \ldots n_m)$):

- Source: $((n_0, n_1, \ldots, n_m), \mathrm{mean}((n_0, n_1, \ldots, n_m)))$
- Following: $(L_2, \mathrm{mean}(L_2))$

If the output is different, it means a defect in an implementation of f has been detected. In general, given a source test case t_0, its next test case t_f ("follow-up test case") is obtained by applying an MR to t_0.

Segura et al. [27] report several domains where metamorphic testing has been particularly useful. These include thermodynamics, smart streetlight systems, mesh simplification programs, machine learning classifiers, wireless signal metering, search engines, and the NASA Data Access Toolkit (DAT).

3. Related Work

This section explores related work pertinent to both this study and the techniques employed in the proposed method.

Software testing techniques are analyzed and compared according to the white-box and black-box approaches, indicating the advantages and disadvantages of each approach and at which level of software testing development each is typically employed [28,29]. In the context of test case generation techniques, the number of related research papers has increased in recent years, as has the number of studies on the tools that implement them [30,31]. MBT and metamorphic testing may be highlighted as two of the most widely used, showing good results. In [32], software-testing fundamentals, in general, and MBT,

in particular, are presented. Meanwhile, in [33], MBT is cited as one of the most significant automated black- box techniques. Below, we discuss articles that combine and discuss some of the software testing techniques used in the method proposed in the present study.

3.1. Model-Based Testing and Constraint Solvers to Generate Test Cases

C. Sun, M. Li, J. Jia, and J. Han [34] propose a constraint-based model-driven testing approach for web services. They extend the WSDL (Web Services Description Language) to include constraints related to web service behavior, preforming an empirical study with three real-life applications. Despite being complete, the study is specific to web services, unlike our proposal, which aims to be more generic, using a black-box approach.

V. Vishal, M. Kovacioglu, R. Kherazi, and M. R. Mousavi [35] applied MBT using Spec Explorer to the Image Detection Subsystem responsible for generation, detection and translation of X-rays to images. They developed their own tool that interfaces Spec Explorer with a constraint solver to generate specific test data for the model. They used a small constraint solver called ZogSolve, which is limited in features and has no recent development activity, unlike MiniZinc which provides a graphical interface, supports numerous constraint solvers, and is frequently updated. Furthermore, Spec Explorer is proprietary software, unlike GraphWalker, which is open source.

RESTes [36] is a framework for automated black-box testing of RESTful APIs (representational state transfer application programming interface). Among its main features, RESTest supports the specification and automated analysis of dependencies between parameters, allowing constraint solvers to be used for the automated generation of valid test cases. Similarly to our proposal, it is based on tests for black-box environments, where the code is inaccessible. In this case, open-source software tools are used, such as IDLReasoner. Given an OAS specification and a set of IDL dependencies (e.g., using IDL4OAS), the tool translates them into a CSP expressed in MiniZinc. Subsequently, the results of CSP are analyzed to determine whether an API call satisfies all the dependencies between parameters. Therefore, CSP is used for the same purpose as in the above work by Vishal et al., that is, to manage the dependencies between parameters. However, unlike the work presented here, MBT is used as a technique to obtain the system model, but visual representations, such as those of GW, are not used.

Overall, unlike other approaches, our proposal has the advantage of using two open-source and frequently updated tools. In addition, the proposal aims to be more general, both in applications and in the domains in which it is applied.

3.2. Formalization and Identification of Metamorphic Relations

Various studies have guided or proposed formats for MRs. These studies are briefly described below. Zhang et al. [37] proposed MRI (MR Inferrer), a search-based approach for automatic inference of polynomial MRs for a program under test. In particular, they used a set of parameters to represent a particular class of MRs, which they refer to as polynomial MRs, and turned the problem of inferring MRs into one of searching for suitable values for the parameters. In MRI, the MRs must obey a certain scheme. Furthermore, they are only suitable for numerical values. Additionally, further filtering is necessary to obtain quality MRs. In our proposal, the format of MRs is more flexible, since it is capable of implementing not only polynomial MRs, but all the MRs that can be represented through the constraint programming language (e.g., those including existential/universal quantification or arrays).

Another paper [38] proposes a specific language for describing MRs, called MRDL. This language is descriptive and helps in the implementation of code to generate the following cases from each MR. However, we propose to implement the MRs through a constraint-based language that also implements each MR and obtains the following test cases through a constraint solver.

Segura et al. [27] propose a template for describing MRs, with several examples applicable to different domains and types of software. However, this notation, unlike our proposal, is not sufficiently formal to be processed by a computer and checked for consistency.

In another, more recent work by Segura et al. [39], several ways to represent MRs are proposed, albeit for a specific domain, namely, query languages such as SQL. Therefore, the proposal is not sufficiently homogeneous to be applied to different domains, as proposed in this paper.

Chen et al. [40] propose a methodology called METRIC for the systematic identification of MRs based on a category-choice framework, which partially automates the identification of MRs by obtaining candidate pairs of *test frames* and asking humans to manually identify whether the pair can be used to produce an MR. Test frames are produced by partitioning the input domain, with the approach being supported by a tool called MR-GEN. In some ways, this can be considered a black-box approach to designing MRs, where only the inputs and outputs are important. Sun et al. recently proposed an improved version of METRIC, called METRIC+ [41], which also performs partitioning on the output domain.

To summarize, these works only provide partial guidance on how to identify the MRs and/or generate tests from them. CMBMeTest guides the entire process, from the generation of the source test suite (through MBT+CP) and identification of initial MRs to the generation of follow-up test cases. Moreover, CMBMeTest formalizes the source test cases and the MRs in a constraint programming language for which mature solvers exist.

3.3. Constraint Solving and MT

Constraint solving and metamorphic testing have previously been combined in different ways However, the existing works focus on different aspects, and their approaches are unlike ours. Firstly, Gotlieb and Botella [42] use constraint logic programming to find test cases that do not satisfy a given MR. They present examples where the program can be proved to satisfy this MR. The paper presents a prototype implementation of its approach, using the existing INKA test case generator [43].

Furthermore, a recently published paper by Akgün et al. [44] proposes using MT to validate constraint-programming systems. The complexity of these systems makes them difficult to test otherwise. Specifically, they use MT to validate the Minion CPS [22]. They show that MT is effective in finding bugs introduced through random code mutation. The aim of their work differs greatly from that of CMBMeTest, which uses constraint solving to generate tests for a system under test.

4. Proposed Method

As mentioned, this paper presents CMBMeTest, a method to generate test suites through MBT+CP and MT for software under test, focusing on the design and implementation of MRs. CMBMeTest aims to provide an end-to-end process: in the first steps, it combines model-based testing and constraint programming to derive an initial set of test cases. Constraint programming is also used in later steps to guide the design, implementation and execution of MRs.

This section describes the steps involved in CMBMeTest. Each step is explained in the following subsections. Figure 1 outlines the proposal. Broadly, the methodology consists of the following stages:

1. Gathering information about the problem.
2. Gathering information about the software under test.
3. Source test suite generation.
4. Creation of constraint sets.
5. Implementation of MRs in MiniZinc.
6. Iterative test suite generation and refinement.

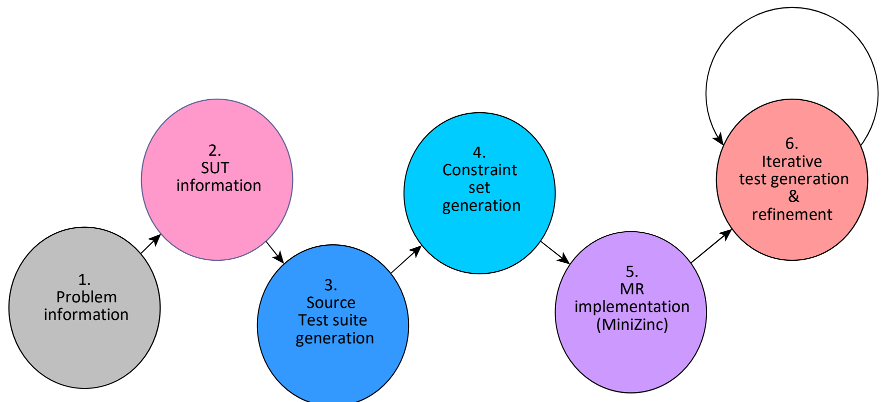

Figure 1. CMBMeTest diagram.

4.1. Gathering Information about the Problem

When using this method, the context in which a problem arises is important because contexts may have different features. The programs that solve the problem can be tested to find potential inconsistencies or flaws by determining the essential components of the problem and how they relate to one another. The procedures to be implemented to determine the context are as follows:

1. Identification of the problem domain or context.
2. Identification of the inputs and outputs for the problem.
3. Identification of the restrictions for the inputs and outputs. That is, the limits of the problem domain in which we can operate. In this step, we write down the nature and number of values of the data we have in the problem, as well as any assumptions. For example, if we want to calculate the average of some given values, we need to know the number of values and whether they are natural numbers, integers, real numbers, etc.
4. Identification of properties related to the problem that affect inputs, outputs or the problem in general. Returning to the example of computing averages, we know that the average should remain the same regardless of the order of the values. That is, even if we permute the order of the values, it should give the same result.

In conclusion, this step is about describing the domain of the problem. If any features or properties are known, they should be included to facilitate the following steps. Section 5.1 shows an example.

4.2. Gathering Information about the Software under Test

This step involves gathering all the information available about the SUT (the input and output value domains are typically defined based on the type of programming language used in the SUT):

1. Specification (format) of the input data, with its program-specific restrictions.
2. Specification (format) of the output data, with its program-specific restrictions.

To develop this proposal, we consider a black-box approach in which the source code is not accessible. Section 5.2 shows an example of this stage.

4.3. Source Test Suite Generation

This stage creates a source test suite according to a set of requirements and to the information gathered in the previous steps, developing a model depicted by a diagram that represents the software under test (SUT).

The method follows a black-box approach, where we explore the input space based on that model, describing the values of the inputs in the paths. We start by constructing a finite

state machine (FSM) model (e.g., in GraphWalker) with an initial state and an `exit` state (marking the completion of a test case). The transitions of the FSM represent the various changes that can be made to the input to the program, and how these affect our expectations of the outputs. Transition guards limit when these changes can be made, and transition actions update two sets: a set of *input requirements* and a set of *output expectations*. Some transitions may only make sense in certain contexts: these contexts are represented as additional states between the initial and `exit` state. The FSM also includes a transition from the `exit` state back to the initial state, whose action outputs a complete test case and clears the input requirements and output expectations, in preparation for constructing the next test case. This looping transition allows for a randomized walk over the graph (meeting a certain stopping criterion, such as covering all edges) to produce requirements for a set of test cases.

The input requirements and output expectations are formalized as sets of parameterized propositions (e.g., `age_greater_than(3)`). A mapping needs to be defined from these propositions to fragments in our chosen constraint programming language (e.g., MiniZinc), which includes constraints and potentially additional variables. For instance, the above proposition would be mapped to `age > 3` in MiniZinc, assuming the `age` variable was defined elsewhere. This use of parameterized propositions is intended to decouple the FSM from the CP language and make it easier to write logic that changes input requirements or output expectations within the action of a transition.

The tests generated by the FSM are then translated to a MiniZinc model in order to generate a test case for the source test suite. In the following, we generically detail the structure of the MiniZinc model that generates each test case:

1. Define the variables corresponding to the inputs and outputs of the test cases in MiniZinc language, according to the type and structure described in the previous steps.
2. Add the constraints included in the diagram for that specific test case (obtained from the GraphWalker path).
3. Add the MiniZinc statement *solve satisfy* so that the solver obtains the values of a test case according to the defined restrictions.

Hence, this step can be automated by translating the set of paths obtained through the diagram into MiniZinc constraint models, as described above. Following the above steps, a source test suite covering the depicted model diagram is obtained. An example can be found in Section 5.3.

4.4. Creation of Constraint Sets

Hereafter, we create sets of restrictions (which will lead to MRs), grouped according to the criteria for obtaining them. This involves making changes in the input variables that impact the output variables and/or other input variables. This requires defining constraints on the input variables before the change (if any), and constraints on the input and output variables after the change. We describe the MRs based on the properties identified in Step 1 of the method, and taking into account the above. For example, if our program calculates the distances between two cities A and B (inputs), with D being the distance (output), and describes a property in which the distance between a source city and a destination city coincides with the distance between the destination city and the source city, an MR can be that, given the source test case t = (A,B,D), the follow-up test case is t′ = (B,A,D′), where D = D′.

Depending on the type of value involved, it may be a simple arithmetic operation. If it is an integer, for example, it may be a deactivation of the variable if it is binary, or simply a substitution of its value by the empty string, if it is a character string, for example. All this has an impact on the output values, and/or on other input variables.

We consider the set of values from the initial test suite as set $T = \{t_1, t_2, \dots t_n\}$, where each $t_i = (I_i, O_i)$ is a specific initial test case with its inputs I_i and its expected outputs O_i. Additionally, we consider the set of values of the follow-up test case as $T' = \{t'_1, t'_2, \dots t'_n\}$, where each $t'_i = (I'_i, O'_i)$ is a follow-up test case obtained from t_i by applying an MR.

The aim is to define several MRs in this way, based on the properties defined above and combining the different changes in the inputs that affect the outputs. Section 5.4 illustrates an example.

4.5. Implementation of Metamorphic Relations in MiniZinc

This section explains how to go from the abstract definition of MRs (as sets of restrictions) in the previous section to their corresponding implementation in MiniZinc. Each MR is implemented as a MiniZinc model. The general steps are as follows:

1. The elements of the initial test case (t_1) must be defined as parameters to be set by the user. Each .dzn file is one of the source test cases obtained in the previous step.
2. The elements of the next test case (t'_1) must be defined as variables to be computed by the solver.
3. A module with constraints may be defined for the source test case.
4. A module with the constraints for the follow-up test is defined.
5. The statement `solve satisfy;` is added, asking for any solution.

Section 5.5 illustrates this mapping for the selected case study.

4.6. Iterative Test Generation and Refinement

With an initial diagram and set of MRs designed, this last stage consists of evaluating and improving both the tests and the MRs in a feedback loop. Specifically, these steps below are followed:

1. Determine the method or technique to be used to evaluate the tests and set the targets to be achieved. For example, use another testing technique (e.g., mutation testing, where common bugs are artificially injected into the program) to assess the quality of the test suite. In the case of mutation testing, the target might be to reach a specific mutation score (i.e., proportion of injected faults that were detected).
2. Assess the source tests generated by MiniZinc based on the chosen technique.
3. Generate the follow-up tests through the MRs implemented and evaluate them, obtaining an indication of how much they have improved.
4. If the target has been achieved, stop.
5. If the target has not been achieved and there has been no improvement in the last repetitions, stop.
6. Otherwise,
    - Redesign the initial diagram with more details (i.e, new states and/or transitions), and input variables (paying attention to the parts of the graph that have not been covered before, if any), and go back to Section 4.3 to generate new test suites from the chosen tool. New additional MRs can likewise be designed.

Section 5.6 presents an example of this stage.

5. Case Study and Evaluation

After presenting the CMBMeTest method with all its steps, this section will set out an end-to-end case study of a system where the source code is not available—in other words, a black-box system. In this case study, the CMBMeTest method is followed, based on the program model obtained from the available information (inputs, outputs, description and specification of the problem and the software used). Using GraphWalker and an FSM, we model the input space of the program and our expectations about the outputs.

The MiniZinc constraint programming system is used to both generate the initial tests, and produce follow-up tests through the identified metamorphic relations.

The case study is based on MetaSearch composition by Mayer and Lübke [9]. Given the two search engines (Google and MSN) and a client that searches for an expression, the program computes the search and obtains the results. In this case, its flowchart is known: both inputs and outputs are specified.

5.1. Gathering Information about the Problem

The problem of search information by Google or MSN would be analyzed as follows:

1. Identification of the problem domain or context. On a general level, this is an orchestration problem of two services that provide equivalent and complementary functionality, where we want to combine their results. In this case, the composition agents are identified. This is a problem of execution of two search engines. We can find three different agents in this composition:
   - The client, who is the one who performs the search query.
   - Google, which is one of the search engines.
   - MSN, which is the second search engine.

   Essentially, a user performs a query using two search engines and obtains a set of results.
2. Identification of the inputs and outputs for the problem.
   - Inputs: Search string entered by the user: questions, language, country, Google results, Google availability, MSN results, MSN availability and maximum number of results.
     - The user provides initial data so that the respective search engines can start the search. (Query, Language, Country, Max_Results).
     - In Google: (url1, title1, snippet1, url2, title2, snippet2, url3, title3, snippet3, . . .). Three strings for each search result, identifying the url, the title and a snippet of the result.
     - In MSN: (title1, description1, url1, title2, description2, url2, url2, title3, description3, url3). Three strings for each search result, identifying the title, description and url.
     - Google availability (binary value).
     - MSN availability (binary value).
     - Max results (int value)
   - Outputs: results between the two search engines according to the logic of the composition and the data provided by the user (maximum number of results).
3. Identification of the restrictions for the inputs and outputs (or both).
   - Input:
     - (Query, Language, Country, Max_Results): value list (alphabetic and numeric) fixed (query, language, country, maximum number of results) (string, string (default), string (default), positive integer or zero)
     - The question strings are open (query), but the language and country strings correspond to the standard language and country abbreviations (ISO 639-2 language code and ISO 3166 country code). For example, es, en, US, GB, ES, etc.
     - Google results: Lists first all Google results in the following format: (url1, title1, snippet1, url2, title2, snippet2, url3, title3, snippet3). (string, string, string, string, string, string, string, . . .) three strings for each search result.
     - MSN results: (Results - GoogleResults $>$ 0) all MSN results will be shown to the user after the possible Google results. (title_msn1, description1, url_msn1, title_msn2, description2, url_msn2, title_msn3, description3, url_msn3). (string, string, string, string, string, string, string, . . .) three strings for each search result.
     - Google availability (binary value: true/false).
     - MSN availability (binary value: true/false).
     - Maximum results (int value)
   - Output: The maximum number of results is determined by the value of MaxResult (number entered by the client), but there can be fewer. This is given the name of expected result.

4. Writing down properties related to the context of the problem that affects inputs, outputs, or the problem in general. These can indicate the impact that changing the inputs in a certain way would have on the program outputs, allowing us to move between different points of the input space of the problem.
   - If none of the search engines are available, the expected result must be 0.
   - If one of the search engines is not available, there are no results from that search engine.
   - If the language value appears, the country value must also appear (and vice versa). The default value is `en-US`.
   - If the number of different results r returned by a search engine is reduced by n for the same search string, the expected results should match the first $r - n$ results of the original search.
   - If a new (different) result is added to one of the search engines, a new expected result should be returned.

5.2. Gathering Information about the Software under Test

The next step is to study the specific characteristics of the program under test that solves the above problem. In the case of MetaSearch, we know that it is a service composition written in WS-BPEL: an XML-based language designed for centralized control of the invocation of different web services, with some added business logic that aids programming in the large. However, we apply the method under a black-box approach, so we assume we do not have the code. We only consider the type of software, and the information available in the problem specification.

1. Specification (format) of input data (restrictions on input information).
   - List of four element tuples (three strings and an integer $>= 0$): `Query`, `Language`, `Country`, `MaxResults`.
     - `Query` is a free string to search.
     - `Language` specifies the language with two lowercase letters (ISO 639-2 language code).
     - `Country` specifies the country with two uppercase letters (ISO 3166 country code).
     - `MaxResults` specifies the maximum results with an integer.
   - There are two types of results, those of Google and those of MSN:
     - `googleAvailable`. Indicates whether the Google search engine is available or not ('true' or 'false').
     - `msnAvailable`. Indicates whether the MSN search engine is available or not ('true' or 'false').
     - `Googlevalues`. Google results are displayed in the following format: (url1, title1, snippet1, url2, title2, snippet2, url3, title3, snippet3). "urlN" follows the format of The URL Standard. (string, string, string, string, string, string, . . .), three strings per search result.
     - `msnResponseValue`. If there are any MSN results (Results - Google_Results > 0), all MSN results will be shown to the user after the possible Google results: (title_msn1, description1, url_msn1, title_msn2, description2, url_msn2, title_msn3, description3, url_msn3) (string, string, string, string, string, string, . . .), three strings per search result. "url_msn$N$" follows the format of the URL Standard.
2. Specification (format) of the output data (restrictions on output information).

5.3. Source Test Suite Generation

We have divided this phase into two steps: on the one hand, the creation of the diagram in GraphWalker that will generate coverage paths, and on the other hand, the generation of test cases from the paths obtained and using the constraint model in MiniZinc.

Based on the information about MetaSearch obtained in the previous steps, a diagram is developed in GraphWalker that represents the model of the software under test (Figure 2).

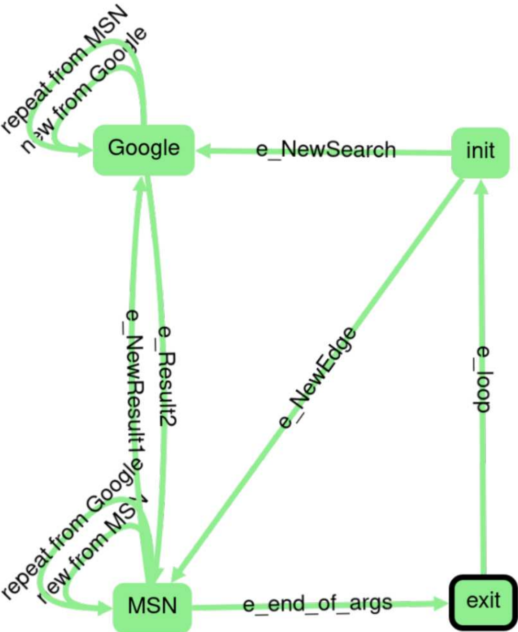

Figure 2. MetaSearch initial GraphWalker diagram, GW1.

For the creation of the diagram, four states are defined: `init` (initial state), `exit` (exit state, marking the completion of a test case), `Google` (represents the Google search engine) and `MSN` (represents the MSN search engine). Furthermore, the transitions in the diagram in Figure 2 are described below, including the source and destination states and what they represent:

- `e_NewSearch`: goes from `init` to `Google`. It models a result in the Google search engine.
- `e_NewEdge`: goes from `init` to `MSN`. It models a result in the MSN search engine.
- `e_loop`: goes from `exit` to `init`. It models the start of a new path when another path has finished. Its action outputs a complete test case and clears the input requirements and output expectations in preparation for constructing the next test case. This looping transition allows for a randomized walk over the graph.
- `new from Google`: goes from `Google` to `Google`. It models when having received a result from Google, you obtain a new result from Google.
- `repeat from MSN`: goes from `Google` to `Google`. It models when the result from Google is a repeated result from MSN.
- `e_NewResult1`: goes from `MSN` to `Google`. It models when a result arrives from Google having previously received a result from MSN.
- `e_Result2`: goes from Google to `MSN`. It models when a result arrives from MSN having previously received a result from Google.
- `new from MSN`: goes from `MSN` to `MSN`. It models when having received a result from MSN, a new result is obtained from MSN.
- `repeat from Google`: goes from `MSN` to `MSN`. It models when the result from MSN is a repeated result from Google.
- `e_end_of_args`: goes from `MSN` to `exit`. It models the end of a path that the diagram has covered.

Some guards and actions are included in the transitions and states. For example, as a guard, the maximum number of results for each search engine is 10. In the actions, the input conditions and the output conditions of each transition are included.

Furthermore, as a feature of the generator, we specify *random(edge_coverage(100) && reached_vertex(exit))*. This performs a random walk, which stops when all edges are covered and the `exit` state is reached.

In this first version of the diagram, for simplicity, it is considered that both search engines are always available (`google_av` = true and `MSN_available` = true), and that the values of `language` and `country` are fixed.

Invoking GraphWalker produces a set of paths that meets the stated restrictions and achieves the required coverage. To create a test case for the source test suite, a MiniZinc model is used to represent each of these paths. Each guard and each input and output condition (expressed as a logic proposition, e.g., "expected_result_count(n)") of the GraphWalker diagram are translated into a MiniZinc constraint. An excerpt of the propositions and how they map to MiniZinc can be seen in Table 1. Therefore, for example, the first row of Table 1, *at_ least_nresults_msn(n)*, as an input constraint, means that there are at least *n* results from the MSN search engine. This sentence is translated into MiniZinc language by declaring an integer variable called `num MSN` and adding the constraint that its value must be ≥`n`.

Table 1. Excerpt of mappings from intermediate code (logical propositions from GraphWalker model) examples to MiniZinc constraints.

Proposition	MiniZinc Constraint
at_least_nresults_msn(n)	`var int: num_MSN; num_MSN >= n;`
at_least_nresults_google(n)	`var int: num_G; num_G >= n;`
expected_result_count(n)	`var int: num_expected; num_expected = n;`

Below, we detail the particular steps of the MiniZinc model structure that generate each test case for MetaSearch:

1. Define the variables corresponding to the inputs and outputs of the test cases in MiniZinc language

```
var int: num_MSN; % number of MSN results
var int: num_G; % number of Google results
var int: num_expected; % number of results expected
array[1..10] of var string: listG; % Google results
array[1..10] of var string: listMSN; % MSN results
```

2. Add the constraints included in the diagram for general test cases and specific test cases. An example of one of the paths, where both Google and MSN results can be ≥ 0, but the number of expected results is 0:

```
%% The maximum number of results is limited to 10:
array[1..10] of var string: listG; % Google results
array[1..10] of var string: listMSN; % MSN results
constraint forall(i in 1..10)
(exists(j in 1..21)(listG[i] = listURLs[j]));
constraint forall(i in 1..10)
(exists(j in 1..21)(listMSN[i] = listURLs[j]));
constraint num_MSN >= 0;
constraint num_G >= 0;
constraint num_expected = 0;
```

3. Add the MiniZinc statement *solve satisfy* so that the solver obtains the values of a test case, according to the defined restrictions.

Listing 2 implements Path 1, which corresponds to "no results from any search engine". Running this MiniZinc model generates tc1, which represents Test Case 1. Therefore, this process can be automated by converting the collection of paths acquired using GraphWalker into MiniZinc constraint models, as mentioned. Repeating this process over all the paths generated by GraphWalker produces a test suite that can be used to exercise the system under test.

Listing 2. Sample MiniZinc model of Path 1 (tc1 generation).

```
1  %% Path 1: No results from any search engine
2  %% CONSTANTS
3  array [1..21] of string: listURLs;
4  listURLs = ['' https://www.uca.es'', ''https://webmerlin.uca.es'', '' https://www.
       uma.es'', ''https://www.ugr.es'', ''https://www.ucm.es'', ''https://www.upm
       .es'', ''https://www.upc3m.es'', ''https://www.uhu.es'', ''https://www.ujaen.
       es'', ''https://www.uco.es'', ''https://www.upo.es'', ''https://www.us.es'', ''
       https://www.usc.gal'', ''https://www.upc.edu'', ''https://www.unex.es'', ''
       https://www.uva.es'', ''https://www.uclm.es'', ''https://www.usal.es'', ''
       https://www.uoc.edu'', ''https://www.upv.es'', ''https://www.uniovi.es''];
5  % VARIABLES
6  var int: num_MSN; % number of MSN results
7  var int: num_G; %number of Google results
8  var int: num_expected; %number of results expected
9  array [1..10] of var string: listG; %Google results
10 array [1..10] of var string: listMSN; %MSN~results
11
12 %% GENERAL CONSTRAINTS
13 %% The maximum number of results is limited to 10:
14 constraint forall(i in 1..10) ( exists(j in 1..21) (listG[i] = listURLs[j]));
15 constraint forall(i in 1..10) ( exists(j in 1..21) (listMSN[i] = listURLs[j]));
16
17 %% PATH-SPECIFIC CONSTRAINTS
18 % input conditions: at_least_nresults_msn (0), at_least_nresults_google (0)
19 constraint num_MSN >= 0;
20 constraint num_G >= 0;
21 % output conditions: expected_result_count (0)
22 constraint num_expected = 0;
23 %% SOLUTION
24 solve satisfy;
```

5.4. Creation of Constraint Sets

In this section, the MRs are designed according to the restrictions and conditions of input, output and paths. We use the following notation to describe MRs, based on inference rules:

$$\text{RULE NAME}\ \frac{\text{premises}}{\text{conclusion}}$$

In the above example, *premises* is a list of predicates that must hold before the MR can be applied to a test (with an implicit logical AND between them), and *conclusion* is another list of predicates that will hold on the follow-up test case (also with an implicit logical AND). We use these variables to describe the various parts of each test case:

- $\mathrm{SE_eAv_t}$ is a boolean variable which is true if search engine $e \in \mathbb{N}^+$ is available on test t, where $t \in \{s, f\}$ (where s refers to the source test case, and f refers to the follow-up test case).

- $\mathrm{SE_eResult_t}$ is an integer variable that represents the number of results from engine $e \in \mathbb{N}^+$ on test t, where $t \in \{s, f\}$ (where s refers to the source test case, and f refers to the follow-up test case).
- $\mathrm{num_expected_f}$ is an integer variable that represents the number of total results expected from both engines.
- m is an integer value > 0 that allows a variable to be decremented or incremented.
- $\mathrm{Country_t}$ is a string variable that represents the country symbol on test t, where $t \in \{s, f\}$ (where s refers to the source test case, and f refers to the follow-up test case). When the value is missing, the default value is *US*.
- $\mathrm{Language_t}$ is a string variable that represents the language symbol on test t, where $t \in \{s, f\}$ (where s refers to the source test case, and f refers to the follow-up test case). When the value is missing, the default value is *en*.

Items marked with s represent the source test case (t_s) and items marked with f represent the corresponding follow-up test case (t_f).

For the specification and design of the MRs, we consider the changes in the availability of search engines ($\mathrm{SE_eAv_t}$) and increase or decrease in the number of results ($\mathrm{SE_eResult_t}$) for each search engine, forgetting the language ($\mathrm{Language_t}$) or country ($\mathrm{Country_t}$). All these changes entail changes in some of the follow-up values ($\mathrm{num_expected_f}$, $\mathrm{Country_f}$ or $\mathrm{Language_f}$, for example).

The following MRs (MR1_MSAv and MR2_MSAv) represent when one of the search engines is no longer available, while the other engine is available. In that case, therefore, the engine goes from being available to being unavailable and from having a number of results to having 0 results. The total number of expected results must match the results of the other search engine. Likewise, MR3_MSAv represents when both of the search engines are no longer available. Therefore, the total number of expected results must match with 0.

MRn_MSAv (a search engine ($\mathrm{SE_e}$) becomes unavailable or all search engines become unavailable):

$$\textbf{MR1_MSAv}\ \frac{\mathrm{SE_1Av_s} \qquad \mathrm{SE_2Av_s}}{\begin{array}{c}\neg\mathrm{SE_1Av_f} \qquad \mathrm{SE_2Av_f} \qquad \mathrm{SE_1Result_f} = 0 \\ \mathrm{SE_2Result_f} = \mathrm{SE_2Result_s} \qquad \mathrm{num_expected_f} = \mathrm{SE_2Result_s}\end{array}}$$

$$\textbf{MR2_MSAv}\ \frac{\mathrm{SE_1Av_s} \qquad \mathrm{SE_2Av_s}}{\begin{array}{c}\mathrm{SE_1Av_f} \qquad \neg\mathrm{SE_2Av_f} \qquad \mathrm{SE_2Result_f} = 0 \\ \mathrm{SE_1Result_f} = \mathrm{SE_1Result_s} \qquad \mathrm{num_expected_f} = \mathrm{SE_1Result_s}\end{array}}$$

$$\textbf{MR3_MSAv}\ \frac{\mathrm{SE_1Av_s} \qquad \mathrm{SE_2Av_s}}{\begin{array}{c}\neg\mathrm{SE_1Av_f} \qquad \neg\mathrm{SE_2Av_f} \qquad \mathrm{SE_1Result_f} = 0 \qquad \mathrm{SE_2Result_f} = 0 \\ \mathrm{num_expected_f} = 0\end{array}}$$

The following MRs (MR1_MSSub and MR2_MSSub) represent when the number of different results from one of the search engines ($\mathrm{SE_eResult_t}$) is decremented by m. Consequently, the number of expected results is likewise decreased accordingly ($\mathrm{num_expected_t}$).

MRn_MSSub (decrease by m the number of different results of a search engine):

$$\textbf{MR1_MSSub}\ \frac{\mathrm{SE_1Result_s} > 0 \qquad \mathrm{SE_2Result_s} >= 0}{\begin{array}{c}\mathrm{SE_1Result_f} = \mathrm{SE_1Result_s} - m \qquad \mathrm{SE_2Result_f} = \mathrm{SE_2Result_s} \\ \mathrm{num_expected_f} = \mathrm{num_expected_s} - m\end{array}}$$

$$\textbf{MR2_MSSub}\ \frac{\mathrm{SE_1Result_s} >= 0 \qquad \mathrm{SE_2Result_s} > 0}{\begin{array}{c}\mathrm{SE_1Result_f} = \mathrm{SE_1Result_s} \qquad \mathrm{SE_2Result_f} = \mathrm{SE_2Result_s} - m \\ \mathrm{num_expected_f} = \mathrm{num_expected_s} - m\end{array}}$$

The following MRs (MR1_MSAdd and MR2_MSAdd) represent when the number of different results from one of the search engines ($SE_eResult_t$) is incremented by m. Consequently, the number of expected results is likewise incremented accordingly ($num_expected_t$).

MRn_MSAdd (increase by m the number of results of a search engine):

$$\textbf{MR1_MSADD}\ \frac{SE_1Result_s >= 0 \qquad SE_2Result_s >= 0}{\begin{array}{c} SE_1Result_f = SE_1Result_s + m \qquad SE_2Result_f = SE_2Result_s \\ num_expected_f = num_expected_s + m \end{array}}$$

$$\textbf{MR2_MSADD}\ \frac{SE_1Result_s >= 0 \qquad SE_2Result_s >= 0}{\begin{array}{c} SE_1Result_f = SE_1Result_s \qquad SE_2Result_f = SE_2Result_s + m \\ num_expected_f = num_expected_s + m \end{array}}$$

The following MRs (MR1_MSCult, MR2_MSCult and MR3_MSCult) are related to the information called "Culture", which consists of the language symbol and the country symbol. The change, in this case, consists of deleting either the `language` value (MR1_-MSCult), the `country` value (MR2_MSCult) or both (MR3_MSCult). Bearing in mind that when one of the two values is missing, as discussed, the default values *en* (language) and *US* (country) are set, these should be the culture values in the follow-up test case generated.

MRn_MSCult (delete one or both values of Culture: $Language_t$, $Country_t$)

$$\textbf{MR1_MSCULT}\ \frac{Language_s = \text{``''} \qquad Country_s != \text{``''}}{Language_f = en \qquad Country_f = US}$$

$$\textbf{MR1_MSCULT}\ \frac{Language_s != \text{``''} \qquad Country_s = \text{``''}}{Language_f = en \qquad Country_f = US}$$

$$\textbf{MR1_MSCULT}\ \frac{Language_s = \text{``''} \qquad Country_s = \text{``''}}{Language_f = en \qquad Country_f = US}$$

5.5. Implementation of Metamorphic Relations in MiniZinc

This section follows the steps to implement the MRs corresponding to MetaSearch in MiniZinc models, according to the previous section.

1. The elements of the initial test case (t_1) must be defined as parameters to be set by the user:

```
int: num_MSN1;
int: num_G1;
int: num_expected1;
array[1..10] of string: listG1;
array[1..10] of string: listMSN1;
string: av_msn1;
string: av_google1;
string: language1;
string: country1;
```

2. The elements of the next test case (t_2) must be defined as variables to be computed by the solver:

```
var int: num_MSN2;
var int: num_G2;

var int: num_expected2;
```

```
array[1..10] of var string: listG2;
array[1..10] of var string: listMSN2;
var string: av_msn2;
var string: av_google2;
var string: language2;
var string: country2;
```

3. A module with constraints may be defined for the initial test case (for example, MR1_MSCult.mzn, remove the country):

```
%% country1 must be to != "";
constraint country1 != "";
constraint num_MSN1 >= 0;
constraint num_G1 >= 0;
constraint num_expected1 >= 0;
```

4. A module with the constraints for the follow-up test is defined:

```
constraint num_MSN2 = num_MSN1 ;
constraint num_G2 = num_G1;

constraint num_expected2 = num_expected1;
constraint forall(i in 1..num_G1)(listG2[i] = listG1[i]);
constraint forall(i in 1..num_MSN1)(listMSN2[i] = listMSN1[i]);

constraint av_msn2 = av_msn1;
constraint av_google2 = av_google1;
constraint language2 =  language1;
constraint country2 = "" ;
```

5. The statement `solve satisfy;` is added, asking for any solution.

The parameter values for every initial test case are written in a different file with extension `dzn` with the test case name. Indeed, each .dzn file is one of the initial test cases obtained in the previous step. As an example, tc0.dzn is included as a source test case, Listing 3.

Listing 3. Sample MiniZinc source test case (parameters), tc0.dzn

```
1 listG1 = [“ https://www.us.es”, “https://www.us.es”, “https://www.us.es”, “
      https://www.us.es”, “https://www.us.es”, “https://www.us.es”, “https://
      www.us.es”, “https://www.us.es”, “https://www.us.es”, “https://www.us.
      es”];
2 listMSN1 = [“https://www.ucm.es”, “https://www.uca.es”, “https://www.us.es
      ”, “https://www.us.es”, “https://www.us.es”, “https://www.us.es”, “https
      ://www.us.es”, “https://www.us.es”, “https://www.us.es”, “https://www.
      us.es”];
3 num_G1 = 0;
4 num_MSN1 = 3;
5 num_expected1 = 3;
6 av_google1 = “true ”;
7 av_msn1 = “true”;
8 country1 = “DE”;
9 language1 = “de ”;
```

Note that Steps 1 to 5 should be written in the model file .mzn (MiniZinc program file). Table 2 shows the list and description of the MRs implemented for MetaSearch.

Table 2. Metamorphic relations for Metasearch.

Name	Description
MR1_MSAv	Turn off MSN
MR2_MSAv	Turn off Google
MR3_MSAv	Turn off MSN and Google
MR1_MSAdd	Add a new MSN result
MR2_MSAdd	Add a new Google result
MR1_MSSub	Subtract a result from MSN
MR2_MSSub	Subtract a result from Google
MR1_MSCult	Language is removed
MR2_MSCult	Country is removed
MR3_MSCult	Language and Country are removed

5.6. Iterative Test Generation and Refinement

Once the GraphWalker diagram is created, the source test suite of test cases is generated, the MRs are implemented, and the follow-up MetaSearch test cases are generated, this step involves evaluating the results obtained and improving them if necessary. For this purpose, the process is iterated over some of the steps. Specifically, these steps are followed:

1. Determine the method or technique to be used to evaluate the evidence and set the objectives to be achieved. Mutation testing is used in this case study to evaluate the results, using the MuBPEL tool [45] for mutation testing for WS-BPEL programs.
2. Assess the source tests generated by MiniZinc, based on the chosen technique. The source test suite, with seven test cases, is evaluated with MuBPEL, resulting in a percentage in 75% of mutants killed.
3. Generate the follow-up test suite through the implemented MRs and evaluate them, obtaining an indication of how much they have improved. We have implemented 10 MRs (see Table 2), obtaining, in total, 60 follow-up test cases. The percentage of mutants killed was 92%.
4. Although the result is improved by applying MRs (from 75% mutation score to 92%), we seek to improve it by redesigning the GraphWalker diagram, adding more variability in the information and transitions.
    - Redesign the GraphWalker diagram with more details (i.e, new edges or vertexes), and input variables (paying attention to the parts of the graph that have not been covered before, if any), and go back to Section 4.3 to generate new test suites from the GraphWalker tool. The initial GraphWalker diagram is refined by adding a new state (`cultures`) to allow for different values in `Country` and `language`, as well as adding new transitions related to this new state. Additionally, new input conditions are added with information about the availability of each search engine(`av_msn(true/false)` and `av_google(true/false)`). The refined GraphWalker diagram of Metasearch can be observed in Figure 3.

The following steps were followed with the new refined Metasearch diagram, generating a source set of coverage test cases, specifically 29 test cases. Subsequently, MuBPEL was run to obtain the mutation score, obtaining a mutation score of 90%, killing a total of 573 mutants.

Further test cases were generated by applying the 10 previously designed MRs to the initial set. From this operation, 119 test cases were generated, killing a total of 584 mutants, obtaining a mutation score of 91.8%. The results achieved in the complete process are shown in Table 3. The results obtained are very similar, and so, in principle, we consider that the mutation score is improved with respect to the generated source sets. However, a more detailed discussion is presented in the following section.

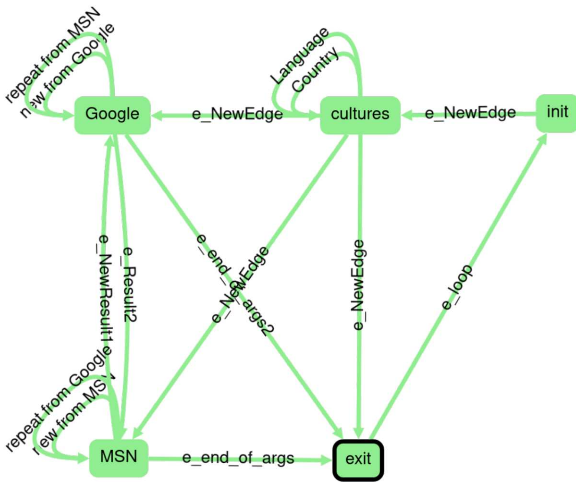

Figure 3. Refined Metasearch GraphWalker diagram.

Table 3. Test cases generated and mutants killed by source and follow-up test suites from initial and refined GraphWalker diagram.

Phase	Source	Generated Tests	Mutants Killed	Mutants Percentage Killed
Round 1	GW1	7	478	75.0%
	GW1 + MRs	60	585	92.0%
Refinement	GWR	29	573	90.0%
	GWR + MRs	119	584	91.8%

As an overview, the process carried out in this case study through this phase is illustrated in the flow diagram in Figure 4.

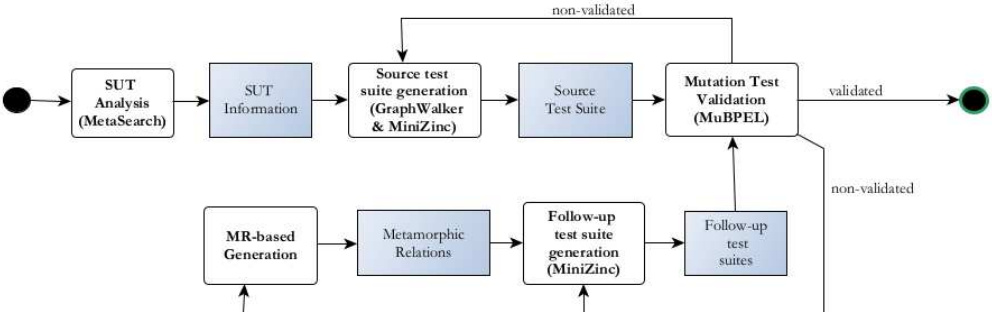

Figure 4. MetaSearch CMBMeTest iterative test generation and refinement phase diagram.

6. Discussion

A proposal was made to generate a coverage test suite using a black-box approach. For this purpose, a step-by-step guide was proposed, from the construction of the system model, the specification, and the available information, represented as a finite state machine, to the evaluation of the obtained coverage test suite, with it being possible to iterate the process in some of the steps.

We propose the integration of several techniques in which MBT is used to obtain the system model that generates paths that ensure coverage and CP is used to obtain the source test suite from those paths. Subsequently, metamorphic testing is used to improve the source test suite by generating follow-up test suites through the execution of MRs. The MRs are implemented as a set of constraints. Finally, to evaluate the result, mutation testing is used.

For the validation of the process, a case study was developed based on MetaSearch, a service composition that orchestrates queries from a client over two search engines: Google

and MSN. The input space of the system under test and the expectations over its outputs were modeled with GraphWalker as finite state machines. GraphWalker provides both a graphical model editor, and a tool to execute the model by performing random walks until a stopping criterion is met.

A first diagram of the system was produced with the basic information (called GW1). The paths were generated, indicating full edge coverage to GraphWalker and including the input and output conditions in the actions and the necessary conditions in the guards. From the path information, MiniZinc models were implemented to generate the source set of test cases. The set used consisted of seven test cases. This initial set of test cases was evaluated with mutation testing using MuBPEL, since MetaSearch is written in WS-BPEL. MuBPEL generated a total of 706 mutants. These mutants were run against the source set of test cases.

The remaining live mutants were manually analyzed to discard equivalent mutants, leaving a total of 636 mutants. When evaluating this source set against the mutants, 478 mutants were killed, indicating a mutation score of 75%. Furthermore, MRs were then specified as constraints on the basis of some of the input data, performing operations that affect the outputs and/or other data in the model. For example, some relations were specified and implemented with respect to the availability of either search engine, or both. If, for example, Google becomes unavailable, there can be no results from that search engine, and the expected results would come only from the other search engine. That is, from a source case in which Google is available, another test case is generated in which Google becomes unavailable. Therefore, the number of results from Google is modified (which would be 0, as it is not available), and the list of results from Google would be the empty list, as well as the expected results. In general, in the case that it is available, there would only be the results from the other search engine . In total, 10 MRs were implemented, generating a total of 60 test cases. We evaluated this follow-up test suite with MuBPEL, killing a total of 585 mutants, indicating a mutation score of 92%. This suggests that the resulting test suite improved considerably.

While the initial diagram did not specify variability in all the input data, the model diagram was refined to include other information, such as language, country, or availability of search engines. This was carried out by adding one more state to the model, several additional transitions and new input and output conditions. A new refined diagram was obtained, which we named GWR. From this diagram, we obtained a set of paths, which were represented in MiniZinc, obtaining a test suite of 29 coverage test cases. This set was evaluated with MuBPEL, and 573 mutants were killed, yielding a mutation score of 90%. It is noted that just by refining the diagram, the mutation score is substantially improved.

From this test suite, we also applied the 10 previously implemented MRs, obtaining a total of 119 test cases below. Subsequently, we evaluated using MubPEL again, killing a total of 484 mutants, indicating a mutation rate of 91.8%. As can be seen, the result obtained is similar to that obtained with the follow-up test suite of the initial diagram (GW1).

Therefore, in each case, the time and effort to be spent on the refinement of the model diagram and the generation of the next test cases must be assessed. In the first case, the number of cases was small, so fewer resources were used in the generation of the diagram, but the mutation score was lower, such that the way to improve it is to obtain a set of subsequent cases with the MRs.

In addition, by refining the diagram, the mutation score was improved to values similar to those obtained with the follow-up test cases of the MRs (90% vs. 92%). It should be taken into account that, in this refinement, a much larger set of source test cases was generated (29 versus 7 from GW1), and, therefore, when applying the MRs, many more follow-up test cases were also generated than in the case of GW1 + MRs (119 versus 60 for GW1 + MRs).

In general, the experience with MiniZinc was positive in this case study, despite the test cases having required complex or different data types from those typically used. The work required using the MiniZinc extension to handle strings, as in Amadini et al. [46].

7. Threats to Validity

The results in Section 5.6 show that our approach can automate the generation of new test cases by reusing an existing constraint solver, and that the follow-up test cases can detect more defects. the results are subject to certain threats to validity. Some of these are due to potential flaws in our study (internal), while others limit the ability to generalize our results to other programs and languages (external).

7.1. Internal Threats

The source test suite was created by MiniZinc (as a first output), ensuring there was one case for each path through the composition. It is possible that selecting different initial values or creating a different number of tests could have produced different results.

Ten MRs were defined by only modifying one of the initial test values in each of them. This made it easier to create simple and understandable MRs, which can be quickly mapped to the MiniZinc language. However, further study would be required to see whether this is the right number of MRs: fewer MRs may have been just as good, or more MRs could have improved results even further. It may be possible to define MRs that subsume several of the MRs presented above.

Using constraint solvers to implement MRs helps validate them against inconsistencies. However, it is still necessary to validate the tests against our domain knowledge and our expectations of the program: the MRs operate on the assumption that the source test is valid. Applying an MR as implemented above to an invalid test case would only produce another invalid test case.

7.2. External Threats

There may be test cases or MRs that cannot be represented with the MiniZinc language as-is. For instance, we did not evaluate the use of MiniZinc for programs that operate on more complex data structures (e.g., stacks or queues). For these cases, it may be necessary to extend MiniZinc, or a different approach may be needed. There are extensions of MiniZinc that implement more complex data types, such as vectors or trees [47,48]. These extensions must, however, be supported by the solver; MiniZinc is just a common constraint programming language.

In this case study, most of the inputs and outputs were known. In other domains and programs, only some of these may be known. In this case, the set of parameters and variables would be limited accordingly, based on the information that was available.

8. Conclusions and Future Work

Achieving test suites with high coverage is one of the most important tasks in software testing. There are currently numerous techniques based on different approaches with varying results depending on various factors and contexts. However, there are scant techniques that generically guide the process of generating coverage test suites in black-box approaches, where code is not available. To address this challenge, this paper presents CMBMeTest, a method to generate coverage test suites, which combines several techniques, namely, MBT, CP and metamorphic testing.

The black-box-based process starts from the SUT information available, using MBT to obtain path coverage of the underlying model. Subsequently, CP is used to generate the source test cases, through the input and output constraints. In order to achieve a more complete test suite, we propose using metamorphic testing. For this, it is necessary to design and implement MRs, which generate the follow-up test suite. For this implementation, CP was used, with the constraint programming language MiniZinc.

For the purpose of evaluating the process, a case study was developed based on a service composition consisting of two search engines, called MetaSearch. GraphWalker, as an MBT tool, was used to build the diagram of the software model and to generate the coverage paths.

The source test suite was designed and implemented using MiniZinc, based on identifying the constraints of the paths through the program that achieve a satisfactory starting level of path coverage. MRs are designed by considering perturbations to the inputs and what changes they cause to the outputs. These MRs are also formalized in MiniZinc, such that the solver can automatically produce the follow-up test suite.

The final step in CMBMeTest is an iterative test suite generation and refinement process, which allows for the integration of a test coverage criterion. For the case study in this paper (MetaSearch), we chose mutation analysis as the criterion, using the MuBPEL tool. This process consists of successive iterations of test suite generation or/and MR design and implementation, terminating when the number of mutants killed converges. In each round, new test suites are generated (starting from the source tests) and are given to all the MRs, producing a new follow-up test suite. We obtained promising results from this method, killing more than 90% non-equivalent mutants of the program under test in two rounds: first with a simpler GraphWalker diagram (GW1), and then with a more complex GraphWalker diagram (GWR). In both cases, the follow-up test suites were generated by executing the MRs, and the results were analyzed using MuBPEL. Based on the results obtained in the case study, it can be concluded that, with a simpler diagram and by applying MRs, similar results to those obtained with a more refined diagram can be achieved. It may therefore not be necessary to refine the diagram, which entails great time and complexity. A summary table of the main contributions of this paper is presented in Table 4.

In terms of future work, we envisage that having the MRs formalized in a constraint programming language will have additional applications. For example, if the solver is unable to find a solution for a candidate MR, even after relaxing the constraints on the initial test case, it will provide the designer with strong evidence that the candidate MR may be a logical contradiction and may thus be rejected. We plan to study the integration of these additional applications to an extended version of CMBMeTest.

Finally, we aim to conduct additional studies on the sensitivity of CMBMeTest to the specific choice of the initial tests (e.g., comparing constraint-based design against random generation). We also plan to evaluate the method on a broader set of programs, and to define MR quality metrics based on the effectiveness of their follow-up tests.

Table 4. Highlights of results.

Main Results and Findings
1. Guided method of generating coverage test suites using MBT, CP and MT.
2. Integration of MBT+CP to generate coverage test suite with GraphWalker (MBT) and MiniZinc (CP) tools.
3. Design and implementation of MRs with CP in MiniZinc for a case study.
4. New enhanced test suites for a case study, obtained through MRs.
5. Evaluation of the test suite generated using mutation testing.
6. Assessment and comparison of the generated suites.
7. Results with the more complex model in the iteration were similar to those of the initial model, and so, for this case, it is not worthwhile to design the more complete.

Author Contributions: All the authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by M.C.d.C.-C., A.G.-D. and I.M.-B. The first draft of the manuscript was written by M.C.d.C.-C. and all the authors commented on the previous versions of the manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partly supported by MCIN/AEI/10.13039/501100011033/ and ERDF (the European Regional Development Fund) A way to do Europe grant number AwESOMe Project PID2021-122215NB-C33.

Data Availability Statement: The datasets generated and analyzed during the current study are available from the corresponding author upon request.

Acknowledgments: We are truly grateful to the four anonymous referees for their constructive comments of our paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Rosaria, S.; Robinson, H. Applying Models in Your Testing Process. *Inf. Softw. Technol.* **2000**, *42*, 815–824. [CrossRef]
2. Olsson, N.; Karl, K. GraphWalker, an Open-Source Model-Based Testing Tool. Available online: https://graphwalker.github.io/. (accessed on 4 November 2023).
3. Núñez, A.; Cañizares, P.C.; Núñez, M.; Hierons, R.M. TEA-Cloud: A formal framework for testing cloud computing systems. *IEEE Trans. Reliab.* **2020**, *70*, 261–284. [CrossRef]
4. Hui, Z.W.; Huang, S.; Chua, C.; Chen, T.Y. Semiautomated Metamorphic Testing Approach for Geographic Information Systems: An Empirical Study. *IEEE Trans. Reliab.* **2019**, *69*, 657–673. [CrossRef]
5. Božić, J. Ontology-based metamorphic testing for chatbots. *Softw. Qual. J.* **2022**, *30*, 227–251. [CrossRef]
6. Morán, J.; Bertolino, A.; de la Riva, C.; Tuya, J. Automatic Testing of Design Faults in MapReduce Applications. *IEEE Trans. Reliab.* **2018**, *67*, 717–732. [CrossRef]
7. Chen, T.Y.; Kuo, F.C.; Liu, H.; Poon, P.L.; Towey, D.; Tse, T.H.; Zhou, Z.Q. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* **2018**, *51*, 4:1–4:27. [CrossRef]
8. de Castro-Cabrera, M.C.; García-Domínguez, A.; Medina-Bulo, I. Using constraint solvers to support metamorphic testing. In Proceedings of the 4th International Workshop on Metamorphic Testing, Montreal, QC, Canada, 26 May 2019 ; Xie, X., Poon, P., Pullum, L.L., Eds.; IEEE: Piscataway, NJ, USA , 2019; pp. 32–39. [CrossRef]
9. Mayer, P.; Lübke, D. Towards a BPEL unit testing framework. In Proceedings of the 2006 Workshop on Testing, Analysis, and Verification of Web Services and Applications, Portland, ME, USA, 17 July 2006; Volume 2006, pp. 33–42. [CrossRef]
10. Beizer, B. *Software Testing Techniques*, 2nd ed.; Van Nostrand Reinhold Co.: New York, NY, USA, 1990.
11. Ammann, P.; Offutt, J. *Introduction to Software Testing*, 2nd ed.; Cambridge University Press: New York, NY, USA, 2016.
12. Utting, M.; Pretschner, A.; Legeard, B. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.* **2012**, *22*, 297–312. [CrossRef]
13. Fraser, G.; Arcuri, A. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* **2014**, *24*, 1–42. [CrossRef]
14. Bernardino, M.; Rodrigues, E.M.; Zorzo, A.F.; Marchezan, L. Systematic mapping study on MBT: Tools and models. *IET Softw.* **2017**, *11*, 141–155. [CrossRef]
15. Jaffar, J.; Maher, M.J. Constraint logic programming: A survey. *J. Log. Program.* **1994**, *19*, 503–581. [CrossRef]
16. Apt, K. *Principles of Constraint Programming*; Cambridge University Press: Cambridge, UK, 2003.
17. Rossi, F.; Van Beek, P.; Walsh, T. *Handbook of Constraint Programming*; Elsevier: Amsterdam, The Netherlands, 2006.
18. Kjellerstrand, H. Comparison of CP Systems—And Counting. 2012. Available online: http://www.it.uu.se/research/group/astra/SweConsNet12/hakan.pdf (accessed on 4 November 2023).
19. Kuchcinski, K.; Szymanek, R. JaCoP—Java Constraint Programming Solver. In Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming, Uppsala, Sweden, 16–20 September 2013.
20. Jussien, N.; Rochart, G.; Lorca, X. Choco: An open source Java constraint programming library. In Proceedings of the CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08), Paris, France, 20–23 May 2008; Trick, L.P.A., Ed.; Springer: Paris, France, 2008.
21. Schulte, C.; Tack, G.; Lagerkvist, M.Z. Modeling and programming with gecode. *Schulte Christ. Tack Guid. Lagerkvist Mikael* **2010**. Available online: https://www.gecode.org/doc-latest/MPG.pdf (accessed on 4 November 2023).
22. Gent, I.P.; Jefferson, C.; Miguel, I. Minion: A fast scalable constraint solver. In Proceedings of the ECAI, Riva del Garda, Italy, 29 August–1 September 2006; Brewka, G., Coradeschi, S., Perini, A., Traverso, P., Eds.; IOS Press: Amsterdam, The Netherlands, 2006; Volume 141, pp. 98–102.
23. Nethercote, N.; Stuckey, P.J.; Becket, R.; Brand, S.; Duck, G.J.; Tack, G. MiniZinc: Towards a Standard CP Modelling Language. In Proceedings of the Principles and Practice of Constraint Programming—CP, Providence, RI, USA, 23–27 September 2007; Bessière, C., Ed.; Springer : Berlin/Heidelberg, Germany, 2007; pp. 529–543.
24. MiniZinc Team. MiniZinc Resources. 2014–2023. Available online: https://www.minizinc.org/resources.html (accessed on 4 November 2023).
25. MiniZinc Team. MiniZinc. 2014–2023. Available online: https://www.minizinc.org/index.html (accessed on 4 November 2023).
26. Chen, T.Y. Metamorphic Testing: A Simple Approach to Alleviate the Oracle Problem. In Proceedings of the 5th IEEE International Symposium on Service Oriented System Engineering, Nanjing, China, 4–5 June 2010; O'Conner, L., Ed.; IEEE Computer Society: Piscataway, NJ, USA, 2010.
27. Segura, S.; Duran, A.; Troya, J.; Cortes, A.R. A Template-Based Approach to Describing Metamorphic Relations. In Proceedings of the 2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET), Buenos Aires, Argentina, 22–22 May 2017; Pullum, L.L., Towey, D., Kanewala, U., Sun, C., Delamaro, M.E., Eds.; pp. 3–9. [CrossRef]

28. Nidhra, S.; Dondeti, J. Black box and white box testing techniques-a literature review. *Int. J. Embed. Syst. Appl. (IJESA)* **2012**, *2*, 29–50. [CrossRef]
29. Verma, A.; Khatana, A.; Chaudhary, S. A comparative study of black box testing and white box testing. *Int. J. Comput. Sci. Eng.* **2017**, *5*, 301–304. [CrossRef]
30. Setiani, N.; Ferdiana, R.; Santosa, P.I.; Hartanto, R. Literature Review on Test Case Generation Approach. In Proceedings of the 2nd International Conference on Software Engineering and Information Management, Bali, Indonesia, 10–13 January 2019.
31. Kumar, G.; Chopra, V.; Gupta, D. Systematic Literature Review in Software Test Data Generation. In *Emerging Trends in Engineering and Management. Computing & Intelligent Systems*; SCRS: Delhi, India , 2023; pp. 91–107. [CrossRef]
32. Schieferdecker, I. Model-Based Testing. *IEEE Softw.* **2012**, *29*, 14–18. [CrossRef]
33. Mariani, L.; Pezzè, M.; Zuddas, D. Chapter Four—Recent Advances in Automatic Black-Box Testing. In *Advances in Computers*; Elsevier: Amsterdam, The Netherlands, 2015; Volume 99, pp. 157–193. [CrossRef]
34. Sun, C.; Li, M.; Jia, J.; Han, J. Constraint-based model-driven testing of web services for behavior conformance. In Proceedings of the International Conference on Service-Oriented Computing, Hangzhou, China, 12–15 November 2018; Springer: Cham, Switzerland, 2018; pp. 543–559.
35. Vishal, V.; Kovacioglu, M.; Kherazi, R.; Mousavi, M.R. Integrating Model-Based and Constraint-Based Testing Using SpecExplorer. In Proceedings of the IEEE 23rd International Symposium on Software Reliability Engineering Workshops, Dallas, TX, USA, 27–30 November 2012; pp. 219–224. [CrossRef]
36. Martin-Lopez, A.; Segura, S.; Ruiz-Cortés, A. RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs. In Proceedings of the Service-Oriented Computing, Virtual Event, 22–25 November 2021; Springer: Cham, Switzerland , 2020; pp. 459–475.
37. Zhang, J.; Chen, J.; Hao, D.; Xiong, Y.; Xie, B.; Zhang, L.; Mei, H. Search-Based Inference of Polynomial Metamorphic Relations. In Proceedings of the ASE '14, 29th ACM/IEEE International Conference on Automated Software Engineering, Västerås, Sweden, 15–19 September 2014; Pei-Breivold, H., Ed.; Association for Computing Machinery: New York , NY, USA, 2014; pp. 701–712. [CrossRef]
38. Sun, C.A.; Wang, G.; Wen, Q.; Towey, D.; Chen, T. MT4WS: An automated metamorphic testing system for web services. *Int. J. High Perform. Comput. Netw.* **2016**, *9*, 104–115. [CrossRef]
39. Segura, S.; Durán, A.; Troya, J.; Ruiz-Cortés, A. Metamorphic Relation Patterns for Query-Based Systems. In Proceedings of the MET '19, 4th International Workshop on Metamorphic Testing, Montreal, QC, Canada, 26 May 2019; Bilof, R.S., Ed.; IEEE Press: Piscataway, NJ, USA, 2019; pp. 24–31. [CrossRef]
40. Chen, T.Y.; Poon, P.L.; Xie, X. METRIC: METamorphic Relation Identification based on the Category-choice framework. *J. Syst. Softw.* **2016**, *116*, 177–190. [CrossRef]
41. Sun, C.; Fu, A.; Poon, P.; Xie, X.; Liu, H.; Chen, T.Y. METRIC+: A Metamorphic Relation Identification Technique Based on Input plus Output Domains. *IEEE Trans. Softw. Eng.* **2019**, *47*, 1764–1785. [CrossRef]
42. Gotlieb, A.; Botella, B. Automated Metamorphic Testing. In Proceedings of the COMPSAC 2003, Hong Kong, China, 30 September–3 October 2003; Titsworth, F.M., Ed.; pp. 34–40.
43. Ingenierie, A. *INKA–V1 User's Manual*; Technical Report; Thales Airborne Systems: Mérignac, France, 2002.
44. Akgün, Ö.; Gent, I.P.; Jefferson, C.; Miguel, I.; Nightingale, P. Metamorphic testing of constraint solvers. In Proceedings of the International Conference on Principles and Practice of Constraint Programming, Lille, France, 27–31 August 2018; Hooker, J., Ed.; Springer: Cham, Switzerland, 2018; pp. 727–736.
45. García-Domínguez, A.; Estero-Botaro, A.; Medina-Bulo, I. MuBPEL: Una herramienta de mutacicón firme para WS-BPEL 2.0. In Proceedings of the XVI JISBD, Almería, Spain, 17–19 September 2012; Calero-Muñoz, C., Ángeles, S.-P., Eds.; 2012.
46. Amadini, R.; Flener, P.; Pearson, J.; Scott, J.D.; Stuckey, P.J.; Tack, G. MiniZinc with Strings. *arXiv* **2016**, arXiv:1608.03650 .
47. Caballero, R.; Stuckey, P.J.; Tenorio-Fornés, A. Two type extensions for the constraint modeling language MiniZinc. *Sci. Comput. Program.* **2015**, *111*, 156–189. [CrossRef]
48. Stuckey, P.J.; Tack, G. Enumerated Types and Type Extensions for MiniZinc. In Proceedings of the Integration of Constraint Programming, Artificial Intelligence, and Operations Research, Los Angeles, CA, USA, 20–23 June 2022; Schaus, P., Ed.; Springer: Cham, Switzerland, 2022; pp. 374–389.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.