METRECoP: a methodology for identifying METamorphic RElations through Constraint Programming

M. Carmen de Castro-Cabrera (maricarmen.decastro@uca.es)
University of Cádiz

Antonio García-Dominguez
SEA research group, EPS, Aston University, United Kingdom

Inmaculada Medina-Bulo
University of Cádiz

Research Article

Keywords: Metamorphic Relations, Constraint Solvers, Constraint Programming Systems, Software Validation

Posted Date: May 16th, 2022

DOI: https://doi.org/10.21203/rs.3.rs-1619280/v1

METRECoP: a methodology for identifying METamorphic RElations through Constraint Programming*

M. Carmen de Castro-Cabrera (0000-0003-4622-5275)[1], Antonio García-Dominguez (0000-0002-4744-9150)[2] and Inmaculada Medina-Bulo (0000-0002-7543-2671)[1]

[1]Department of Computer Science, University of Cádiz, Avda. Universidad de Cádiz, 10, Cádiz, 11519, Spain.
[2]SEA research group, EPS, Aston University, Birmingham, United Kingdom.

Contributing authors: maricarmen.decastro@uca.es; a.garcia-dominguez@aston.ac.uk; inmaculada.medina@uca.es;

Abstract

Metamorphic Testing (MT) is a test design approach that produces "follow-up" test cases from existing ones, through the application of metamorphic relations (MRs). These MRs encode known properties about the problem being solved by the system under test. Identifying, designing and implementing the MRs is costly and error-prone. In the past, several methodologies have been proposed to guide the design of MRs, such as iterative metamorphic testing (IMT) or MR-GEN. In this paper, we propose a refinement of IMT which takes advantage of mature existing constraint programming languages to formalize MRs and automate the generation of test cases. The methodology covers the design of the initial cases, the design of the MRs, and the application of the MRs to produce a comprehensive test suite. METRECoP is applied to an empirical study, creating 10 MRs (by constraint solvers), producing a test suite that can detect all the defects that were seeded into the program, and guiding the design of additional MRs.

Keywords: Metamorphic Relations, Constraint Solvers, Constraint Programming Systems, Software Validation

*Corresponding author: maricarmen.decastro@uca.es

1 Introduction

In the context of software testing, Metamorphic Testing (MT) has proven to be a useful and effective technique in various fields and disciplines, for example, cloud computing (Núñez, Cañizares, Núñez, & Hierons, 2020), geographical information systems (Hui, Huang, Chua, & Chen, 2019), chatbots (Božić, 2022) or distributed computations (Morán, Bertolino, de la Riva, & Tuya, 2018). Metamorphic Testing attempts to alleviate the test oracle problem (Section 2.2) and other problems (improving error detected) through the use of Metamorphic Relations (MRs), which are known relationships between the inputs and outputs of multiple executions of the program under test (Chen et al., 2018). 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 $-y$. With a maturity of twenty years Chen (1998), and several recent major reports that corroborate this Chen et al. (2018); Segura, Fraser, Sanchez, and Ruiz-Cortes (2016), the number of publications per year is increasing, with a dedicated yearly workshop for exchanging advances on the technique from 2016: O'Conner (2021); Xie, Poon, and Pullum (2019); Xie, Pullum, and Poon (2018). However, there are still challenges to be addressed and issues that need to be researched and developed, as highlighted in the recent review by Chen et al. (2018), which pointed out the challenge of "systematic MR identification and selection".

In the literature, several methodologies have been proposed, such as IMT Wu (2005), METRIC Chen, Poon, and Xie (2016), or METRIC+ Sun et al. (2019). These methodologies typically operate in an iterative manner, where an initial set of test cases is generated, and then augmented through repeated applications of MRs. However, they require the manual definition of MR sequences (as in IMT)[1], or they focus on input/output partitioning (as in METRIC/METRIC+) and do not take into account the high-level logic of the program.

Likewise, some machine-parsable notations for describing MRs have been proposed, such as MRDL from Sun, Wang, Wen, Towey, and Chen (2016). The MRDL authors described an initial algorithm for generating test inputs from the descriptions, which relied on heuristics and filtering over a random generator, rather than using an existing constraint solving system.

In this paper, we propose METRECoP (METamorphic RElations through Constraint Programming), a methodology which aims to solve the above issues. METRECoP follows a white-box approach, and uses a generational approach for MRs that does not require manually defining MR sequences. METRECoP uses an existing mature constraint programming language (MiniZinc) to specify MRs, and leverages the available constraint solving systems to compute follow-up tests from the MRs.

[1]By applying metamorphic relations in a chain style. Given a metamorphic relation sequence $MR_1, ..., MR_n$, new test cases resulted from metamorphic relation $MR_i (1 \leq i < n)$ can be reused as source test cases for the next metamorphic relation MR_{i+1}

Following from our prior proof-of-concept MR identification study M.C. de Castro-Cabrera, García-Domínguez, and Medina-Bulo (2019), in this paper we provide a full description of the METRECoP methodology. We apply each step to a Web Services Business Process Execution Language (WS-BPEL) composition inspired by the Triangle program from the Siemens suite (Yang, Khurshid, Person, & Rungta, 2014), and study the relative usefulness of the identified MRs through mutation testing. WS-BPEL has been chosen because there are tools that can evaluate the quality of the tests produced by the methodology via mutation testing, which seeds defects and checks if the tests can detect them. Although the example may seem simple, it is worth noting that most software tools as GIS and terrain studies apply methods based on irregular networks of triangles (TIN).

This paper is structured as follows: Section 2 highlights the background of the subject, Section 3 describes the methodology proposed, Section 4 illustrates and evaluates it, step-by-step through one case study. Section 5 discusses the results answering to the research questions. Section 6 analyzes the threats to validity. Section 7 exposes related works. Finally, the last section presents the conclusions and future lines of research.

2 Background

Before we introduce our methodology, we will present some concepts needed to understand this work. We will start with a general introduction to test case definition and generation approaches, and present metamorphic testing. Then we will move on to the most recent efforts in guiding developers on defining and implementing MRs. Finally, we will introduce constraint programming.

2.1 Test Case Definition and Generation

Beizer (1990) considers that the goal of software testing depends on the maturity level of the software testing process in an organization. He lists 5 levels, of which this paper aims to achieve the last 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 (2016) in one of the basic texts on software testing, the current approach follows two steps: build a model of the program under test (using structures and criteria such as input spaces, graphs, logical expressions, or syntactic representations), and generate tests from this model by finding the various ways in which it can be exercised.

For any program, the possible number of inputs is usually huge, and it is not feasible to test them all at every stage of program development. Therefore, the test designer's goal will be to do a careful sampling of these large input space, expecting to find the least number of tests that show the most number of 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. 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) came about. MDTD decomposes the tests according to the coverage criteria, in 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 could be extracted from decisions in source code, from conditionals in use cases, and so on.

On the other hand, for test design, two approaches should be considered that are complementary (Chapter 2 of Ammann and Offutt (2016)): 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 having experience on 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 (2016) 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, there are many works on how to generate high-quality test cases in an automated way. For instance, Utting et al. discuss in (Utting, Pretschner, & Legeard, 2012) the model-based testing process, defining a taxonomy that covers the key aspects of model-based testing proposals as well as tools to implement them. In addition, Anand et al. review in (Anand, Burke, Chen, et al., 2013) the state of the art on automatic test case generation and updating. Finally, Fraser and Arcuri present in (Fraser & Arcuri, 2014) the results of an experiment on unit test generation using the EvoSuite system on several open source and industrial projects.

2.2 Managing the oracle problem with metamorphic testing

When an output is difficult to verify because the precise result is not known *a priori*,[2] or because the size of the output makes it unfeasible to apply standard techniques, this is called the *oracle problem* (Weyuker, 1980). In these cases, it is recommended to use a different approach to help with the verification and validation process.

An *oracle* can be any mechanism for checking whether a program under test behaves correctly for any given input. Despite the fact that manual checking is error-prone and expensive, it is frequently done. As it is mentioned before,

[2] Partial knowledge or properties of the result can, indeed, be available.

there are practical situations where oracles can be unavailable or too expensive to apply. Metamorphic testing, as mentioned in the previous section, has been proposed to alleviate the oracle problem (Chen, 1998; Chen, 2010). MT relies on the notion of a metamorphic relation (MR). In Chen et al. (2018), 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 $\langle x_1, x_2, ..., x_n \rangle$, where $n \geq 2$, and their corresponding outputs $\langle f(x_1), f(x_2), ..., f(x_n) \rangle$. 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 $R(x_1, x_2, ..., x_n, f(x_1), f(x_2), ..., f(x_n))$ to indicate that $\langle x_1, x_2, ..., x_n, f(x_1), f(x_2), ..., f(x_n) \rangle \in R$.

When the implementation is correct, program inputs and outputs are expected to satisfy some necessary properties that are relevant for 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, $n_0, n_2, n_1, ...n_m$), the result has to be the same, since it is a known property of the arithmetic mean. Formally, we can express this as:

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

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

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

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

Segura et al. mention several domains where metamorphic testing has been particularly useful in Segura, Duran, Troya, and Cortes (2017). Some of these include thermodynamics, smart streetlight systems, mesh simplification programs, machine learning classifiers, wireless signal metering, search engines, or the NASA Data Access Toolkit (DAT).

2.3 Constraint programming

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

This paradigm has its beginnings in the 80s, although there is some prior work. For example, SketchPad from Sutherland (Johnson, 1963) is an interactive constraint-driven drawing tool, and ALICE (Lauriere, 1978) had a generic system of constraints. In the 1990s, they were used more in practice, in general as an extension of logic programming languages: the approach was called Constraint Logical Programming (CLP). Charme (A. Oplobedu & Tourbier, 1989), CHIP V4 (Simonis, 1995) or ILOG (Puget, 1994) are from this stage. They are, in general, software development environments and were used mainly for the development of applications for planning (personnel, production, etc.) and design problems (Jaffar & Maher, 1994).

Constraint Programming Systems (CPSs) are being used more often and in a wider range of domains (Apt, 2003; Rossi, Van Beek, & Walsh, 2006). CPSs are based on different programming languages and paradigms, and many different tools exist. Hakan provided a comparison of the available CPS in Kjellerstrand (2012). 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 (Kuchcinski & Szymanek, 2013), Choco (Jussien, Rochart, & Lorca, 2008), Comet (Van Hentenryck & Michel, 2005), Gecode (Schulte, Tack, & Lagerkvist, 2010), Tailor/Essence (Gent, Jefferson, & Miguel, 2006) , Zinc (Rafeh, 2008), and MiniZinc (Nethercote et al., 2007) were studied. Among 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 plenty of documentation (University & Data61, 2020), iv) it has an active user community around it, and v) it is available as open source.

2.4 MiniZinc

MiniZinc is an open source constraint modeling language (Nethercote et al., 2007) which can describe constraint resolution and optimization problems regardless of the solver used. A graphical interface is included in the default distribution of MiniZinc for ease of use, as well as examples (Data61 research network, CSIRO, 2015). 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. The parameters are set by the user in the data file (file with extension .dzn), and the variables are calculated by the solver while resolving 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 data obtained in the data file, while the language allows flexibility in terms of the operators to apply to that data.

3 Proposed methodology

As mentioned in previous sections, this paper presents METRECoP, a methodology to apply MT to a program under test, focusing on the generation of test cases and the design and implementation of MRs. METRECoP 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 of the methodology proposed: each step will be explained in the following subsections. Figure 1 outlines the methodology. Broadly, the methodology consists of the following stages:

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

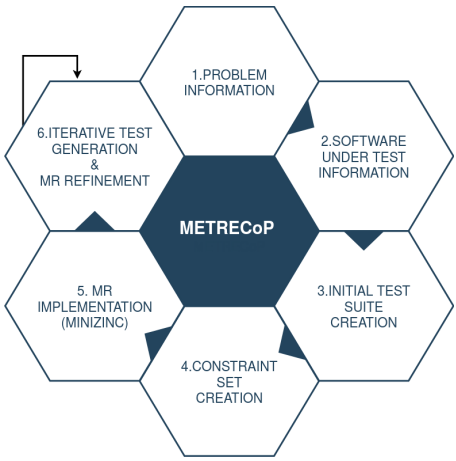

Fig. 1 METRECoP diagram

3.1 Gathering information about the problem

The context in which a problem develops is of interest in the application of this methodology, because there are specific characteristics in different environments. Knowing key elements of the problem and their relationships can help test the programs that solve the problem, detecting possible inconsistencies or errors. Identifying the context requires following these steps:

1. Identification of the name 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 if they are natural numbers, integers, real numbers, etc.
4. Identification of any properties related to the problem that affect inputs, outputs or the problem in general. Going back to the example about computing averages, we know that it should remain the same regardless of the order of the values. That is, 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 ease the following steps. Section 4.1 shows an example.

3.2 Gathering information about the software under test

This step involves gathering all the information available about the software system under test:

1. Programming language used for the implementation, which will impose additional constraints on valid inputs and outputs.
2. Specification (format) of the input data, with its program-specific restrictions.
3. Specification (format) of the output data, with its program-specific restrictions.
4. Available paths through the program, based on known specifications or extracted from the program itself.

For the application of this methodology there are two possibilities: that the source code is accessible and we can apply white-box test design, or that it is not accessible and a black-box approach is needed. In this work, we will focus on the development of an example in which we have the source code and the program logic diagram, where we can apply a white-box approach. Section 4.2 shows an example.

3.3 Creation of the initial test suite

This stage will create an initial set of test cases according to a set of requirements. Those requirements will typically relate to the coverage of the identified paths. If the set of test cases is unwieldy, it is convenient to use one of the test case prioritization (TCP) techniques indicated in M.d.C. de Castro-Cabrera, García-Dominguez, and Medina-Bulo (2020) to sort the set. In principle, other techniques that are more commonly used for the type of problem under consideration could be integrated.

When applying coverage criteria, each of the paths previously identified is translated into one or several constraints in the MiniZinc language. This makes it possible to use a constraint solver to obtain the initial test suite. The result of this step should be a set of test cases that cover the paths identified so far: in the presence of loops, it may be necessary to pick a representative selection. Section 4.3 shows an example.

3.4 Creation of constraint sets

We consider the set of values from the initial test suite as the set $T = \{t_1, t_2, ...t_n\}$, 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. Also, we consider the set of values of the follow-up test case as $T' = \{t'_1, t'_2, ...t'_n\}$, where each $t'_i = (I'_i, O'_i)$ is a follow-up test case obtained from t_i by applying an MR. Following this, we create sets of restrictions (which will lead to MRs), grouped according to different criteria for obtaining them.

We suggest two criteria: one that produces follow-up test cases which stay on the same execution path, and one that produces follow-up tests which move to another path. These criteria guide the design of the conditions that must be met, and the relations between the inputs and outputs of the initial and follow-up tests.

Given an initial test $t_1 = (I, O)$ and a follow-up test $t'_1 = (I', O')$, these MRs can be generally described as:

MRi(S/D) (Starting from the i-path, Stay/move to a Different path, manipulating the initial values):

$$p(t) \wedge f(I, I') \Longrightarrow g(O, O')$$

The predicate p indicates if the MR is applicable for that initial test. The function f relates the inputs of the follow-up test to those of the initial test. Finally, the function g relates the outputs of the follow-up test to those of the initial test. Section 4.4 illustrates an example for the selected case study.

3.5 Implementation of metamorphic relations in MiniZinc

This section explains how to go from the definition of MRs in the previous section in an abstract way to their corresponding implementation in MiniZinc. The general steps are:

1. The elements of the initial test case (t_1) must be defined as parameters to be set by the user.
2. The elements of the next test case (t'_1) must be defined as variables to be computed by the solver.
3. A path-specific constraint module is defined for the initial 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.
6. The parameter values for the initial test case are written in a different file with extension `dzn`, with the same name as the MiniZinc program file.

Note that the steps 1 to 5 should be written in the model file .mzn (MiniZinc program file). Section 4.5 illustrates this mapping for the selected case study.

3.6 Iterative test generation and MR refinement

With an initial 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 are followed:

1. Determine the method or technique to be used to evaluate the evidences and set the objectives to be achieved.
2. Assess the initial tests generated by MiniZinc, based on the chosen technique.
3. Generate the follow-up tests through the implemented MRs and evaluate them, getting an indication of how much they have improved.
4. If the target has not been reached, design new MRs that generate better tests, and go back to step 3. If the result agrees with the set objectives, or there has been no significant improvement after a set number of repetitions, stop.

Section 4.6 shows an example of this stage.

4 Case study and evaluation

After presenting the METRECoP methodology with all its steps, this section will set out several research questions and answer them through an end-to-end case study of a system where the source code is available. The MiniZinc constraint programming system will be used to both generate the initial tests, and produce follow-up tests through the identified metamorphic relations. The research questions are as follows:

1. RQ1: Can a set of metamorphic relations be identified from a selection of execution paths across the source code of a system?
2. RQ2: Can existing constraint programming systems express the extracted metamorphic relations?
3. RQ3: How effective are the test cases generated from the metamorphic relations at detecting defects?

The case study is based on the triangle classification program described in Mcgraw, Michael, and Schatz (1997). Given the lengths of the sides of a triangle (a, b and c), the program computes the type of triangle (equilateral, isosceles, or scalene). In this case, its flowchart is known: inputs and outputs are both specified, as well as the steps to follow to obtain the solution.

4.1 Gathering information about the problem

The problem of identifying the type of triangle would be analysed as follows:

1. Identification of the name of the problem domain or context.

- Flat geometry, triangles, classification according to the length of their sides.

2. Identification of the inputs and outputs for the problem.
   - Inputs: three positive values representing the length of the sides of the triangle: a, b and c.
   - Outputs: "isosceles", "scalene" or "equilateral".
3. Identification of the restrictions for the inputs and outputs (or both).
   - Input: There must be three values greater than 0 to represent the sides of the triangle.
   - Input: The sum of two of its sides must be greater than the size of the remaining side.
   - Output: If these conditions are met, the solution to the problem must be one of these three values or an associated code: "equilateral", "isosceles" or "scalene".
4. Writing down properties related to the context of the problem that affect inputs, outputs, or the problem in general. They are disturbances that allow us to obtain new values or that allow us to move from one point to another in the domain of the problem.
   - Equilateral: if the sides are added or multiplied by the same number, it is still equilateral.
   - Change of order in the sides, the type of triangle must still be kept.

4.2 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. For the case of the triangle, the WS-BPEL implementation from Valle-Gómez (2017) will be used.

1. Write down the programming paradigm and language:
   - The program follows the structured programming paradigm.
   - The program is written in WS-BPEL, a language for writing web service compositions.
2. Specification (format) of input data (restrictions on input information).
   - A list of 3 integer type items (unsigned or signed). When it comes to providing information about the software, it is necessary to specify how the element type is declared. In this case, the definition of a list with a limited length (3) and of integer type.
3. Specification (format) of the output data (restrictions on output information).
   - The output is represented by a numeric code, so that "scalene" is represented by a 1, "isosceles" by a 2 and "equilateral" by a 3.

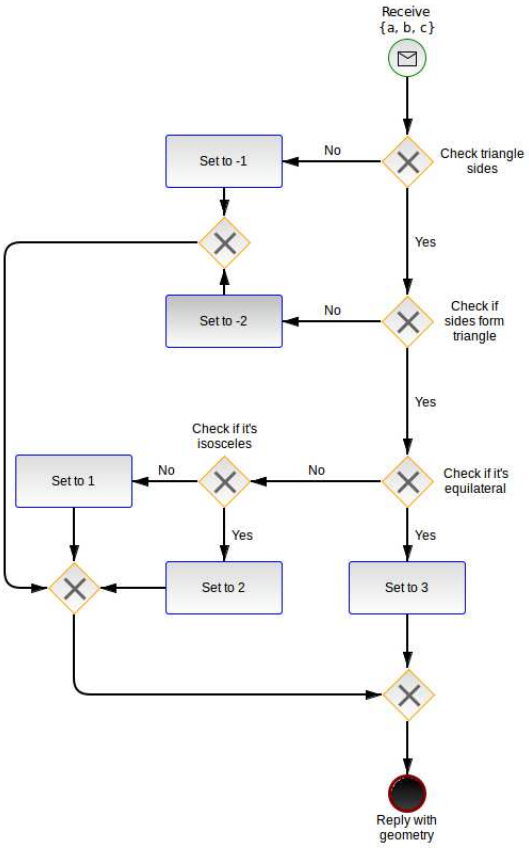

Fig. 2 Triangle diagram

- If one of the lengths is less than or equal to zero, the inputs do not represent a triangle: the output code is -1.
- If the sum of two lengths is not greater than the remaining one, the inputs do not represent a triangle: the output code is a -2.

4. Write down the different paths on the program. According to the diagram in Figure 2, the following testing requirements related to coverage criteria are defined:
   (a) Path 1: there is at least one length less than or equal to zero. The output of this path must be -1.
   (b) Path 2: all lengths are greater than 0 but do not form a triangle. The output of this path must be -2.
   (c) Path 3: all lengths are equal. The output of this path must be 3.
   (d) Path 4: two of the three lengths are equal. The output of this path must be 2.
   (e) Path 5: all lengths are different. The output must be 1.

4.3 Creation of the initial test suite

Considering the testing requirements that cover the paths in Figure 2, which have been described in the previous step, an initial test suite is created. Each test case is a pair $((a, b, c), r)$, where the first element is a list of three values (a, b, c) that corresponds to the sides of the triangle, i.e., the program input; and the second element r is the corresponding program output according to the diagram. To achieve this, input and output values are declared as variables and each path is implemented as a set of constraints in MiniZinc: solving these constraints produces at least one test case per path. The obtained test cases are listed below:

1. Path 1: ((-10000000,1,1), -1).
2. Path 2: ((2,1,1), -2).
3. Path 3: ((1,1,1), 3).
4. Path 4: ((2,2,1), 2).
5. Path 5: ((4,3,2), 1).

As an example, we include the MiniZinc constraints for two of the above paths. Listing 1 implements path 3, which corresponds to an equilateral triangle: as the comment says, while this initial version was written mechanically according to the flowchart, part of it can be simplified away. Listing 2 implements path 5, which corresponds to a scalene triangle.

Once the path constraints are obtained from the diagram, they should be refactored using the structures provided by the constraint language. In some cases, redundant restrictions can be removed (as in the case of path 3).

It can be seen that there is at least one test case for each of the paths, so the flowchart and therefore the code that implements it would be covered.

4.4 Creation of constraint sets

In Section 3.4, two criteria for designing MRs were considered: one which perturbs inputs while staying in the same path, and one which perturbs the inputs to move to a different path. This section will apply both criteria to the current case study.

4.4.1 Follow-up test in same path

These MRs are identified as MRnS, where n is a numeric ID and "S" refers to the fact that the follow-up test belongs to the *(S)*ame path as the initial test case [3].

We observe that for any path, if any two sides are exchanged, the path is maintained. This MR1S could be described in a general way as:

MR1S (Same path, exchange of two values):
$$t_1 = ((a_1, b_1, c_1), r_1) \land\ t_2 = (\text{perm}(a_1, b_1, c_1), r_2) \implies r_2 = r_1$$

[3] Notice that in the development of the case study t_1 represents the initial test case and t_2 the follow-up test case.

```
%% Path 3: all sides are equal, the output must be 3 ( equilateral ).

% VARIABLES
array[1..3] of var int: tuple;
var -2..3: r;

%% PATH-SPECIFIC CONSTRAINTS
% Check all the sides of triangle > 0
constraint forall(l in 1..3) (tuple[l] > 0);
% Check all the sides form a triangle: this restriction is redundant on
    this path, so it can be removed.
constraint tuple[1]+tuple[2] > tuple[3] /\ tuple[1]+tuple[3] > tuple[2]
    /\ tuple[2]+tuple[3] > tuple[1];
% Check all the sides are equal
constraint forall(l in 1..3)
                (forall(m in l+1..3) (tuple[l] = tuple[m]));
% check the property to get a output:
constraint r = 3;

%% SOLUTION
solve satisfy;
output [ "a=", show(tuple[1]), "\n", "b=", show(tuple[2]), "\n", "c=",
    show(tuple[3]), "\n", "r=", show(r), "\n" ];
```

Listing 1 Sample MiniZinc model of Path 3 (equilateral)

```
%% Path 5: all sides are different, the output must be 1 (scalene).

% VARIABLES
array[1..3] of var int: tuple;
var -2..3: r;

%% PATH-SPECIFIC CONSTRAINTS
% Check all the sides of triangle > 0
constraint forall(l in 1..3) (tuple[l] > 0);
% Check all the sides form a triangle:
constraint tuple[1]+tuple[2] > tuple[3] /\
tuple[1]+tuple[3] > tuple[2] /\ tuple[2]+tuple[3] > tuple[1];
% Check all the sides are different
constraint forall(l in 1..3) (forall(m in l+1..3) (tuple[l] != tuple[m]));
% check the property to get a output:
constraint r = 1;

%% SOLUTION (SAME AS LISTING 1)
```

Listing 2 Sample MiniZinc model of Path 5 (scalene)

The function *perm* receives a tuple of 3 elements and returns a tuple with a random permutation of those elements. The MR1S is valid for any path. Below there are some examples of the application of this MR to specific initial tests for each path, and the resulting following test case:

1. Path 1: t_1= ((0,1,2), -1) → t_2= ((0,2,1), -1).
2. Path 2: t_1= ((1,2,1), -2) → t_2= ((1,1,2), -2).
3. Path 3: t_1=((2,2,2), 3) → t_2= ((2,2,2), 3).
4. Path 4: t_1=((2,2,1), 2) → t_2= ((2,1,2), 2).
5. Path 5: t_1=((4,3,2), 1) → t_2= ((4,2,3), 1).

Furthermore, if the three sides of the triangle are multiplied by a positive constant, the type of triangle is also maintained, thus keeping the same path. This MR2S could be described in a general way as:

MR2S (Same path, multiply by a positive constant):
$t_1 = ((a_1, b_1, c_1), r_1) \wedge k > 0 \wedge t_2 = ((k \cdot a_1, k \cdot b_1, k \cdot c_1), r_2) \implies r_2 = r_1$
This MR2S is generic and is suitable for any path. For example, for $k = 3$ and the same values in the initial test suite as in the previous example, we get the follow-up test cases:

1. Path 1: t_1= ((0,1,2), -1) → t_2= ((0,3,6), -1).
2. Path 2: t_1= ((1,2,1), -2) → t_2= ((3,6,3), -2).
3. Path 3: t_1=((2,2,2), 3) → t_2= ((6,6,6), 3).
4. Path 4: t_1=((2,2,1), 2) → t_2= ((6,6,3), 2).
5. Path 5: t_1=((4,3,2), 1) → t_2= ((12,9,6), 1).

4.4.2 Follow-up test in a different path

We identify these MRs as MRnD, where n is the numeric ID for the MR and "D" comes from the fact that the follow-up test case belongs to a *(D)*ifferent path [4]. We observe that, for example, we can make a test run on path 1 by setting some of the lengths to 0. We can define an MR that transforms an initial test case on paths 2/3/4/5 to a follow-up test case on path 1:

MR1D (Change path, set one length to 0):

$$\begin{aligned}
&(t_1 = ((l_1, l_2, l_3), r_1) \\
&\wedge i \in \{1, 2, 3\} \wedge l'_i = 0 \\
&\wedge \forall j \in (\{1, 2, 3\} - \{i\}), l'_j = l_j \\
&\wedge t_2 = ((l'_1, l'_2, l'_3), r_2)) \Longrightarrow r_2 = -1
\end{aligned}$$

The follow-up tests obtained from the relevant initial tests are are described below:

[4]Notice that in the development of the case study t_1 represents the initial test case and t_2 the follow-up test case.

1. Path 2: t_1= ((2,1,1), -2) → Path 1: t_2= ((0,1,1), -1).
2. Path 3: t_1= ((2,2,2), 3) → Path 1: t_2= ((0,2,2), -1).
3. Path 4: t_1= ((2,2,1), 2) → Path 1: t_2= ((0,2,1), -1).
4. Path 5: t_1= ((4,3,2), 1) → Path 1: t_2= ((0,3,2), -1).

We can make a test change to path 3 ("equilateral") by copying one of the lengths (when it is greater than zero) to the other two, in order to obtain an equilateral triangle as the follow-up test case. We can define an MR that turns an initial test case on paths 1/2/4/5 into a follow-up test case on path 3.

MR2D (Change path, a value > 0 is copied to other values):

$$\begin{aligned}&(t_1 = ((a_1, b_1, c_1), r_1)\\ &\wedge \exists x \in \{a_1, b_1, c_1\}, x > 0\\ &\wedge\, t_2 = ((x, x, x), r_2)) \implies r_2 = 3\end{aligned}$$

The results obtained for each path change are described below:

1. Path 1: t_1= ((0,3,2), -1) → Path 3: t_2= ((3,3,3), 3).
2. Path 2: t_1= ((2,1,1), -2) → Path 3: t_2= ((2,2,2), 3).
3. Path 4: t_1= ((2,2,1), 2) → Path 3: t_2= ((2,2,2), 3).
4. Path 5: t_1= ((4,3,2), 1) → Path 3: t_2= ((4,4,4), 3).

A change to path 4 ("isosceles") is obtained by copying one of the lengths (greater than one) into another, when the remaining length is less than twice the value of the other. An MR is then defined that transforms an initial test case on paths 1/2/3/5 into a follow-up test case on path 4.

MR3D (Change path, a value > 1 is copied to another value):

$$\begin{aligned}&t_1 = ((l_1, l_2, l_3), r_1)\\ &\wedge \exists i \in \{1, 2, 3\}, l_i > 1\\ &\wedge \exists j \in (\{1, 2, 3\} - \{i\}), l_j < 2l_i\\ &\wedge\, t_2 = ((l_i, l_i, l_j), r_2) \implies r_2 = 2\end{aligned}$$

The results obtained for each change of path are described below:

1. Path 1: t_1= ((0,3,2),-1) → Path 4: t_2= ((3,3,2), 2).
2. Path 2: t_1= ((2,1,1),-2) → Path 4: t_2= ((2,2,3), 2).
3. Path 3: t_1= ((2,2,2), 2) → Path 4: t_2= ((2,2,3), 2).
4. Path 5: t_1= ((4,3,2), 1) → Path 4: t_2= ((4,4,2), 2).

4.5 Implementation of metamorphic relations in MiniZinc

The MRs defined in the previous section were mapped to MiniZinc models, following the steps below:

```
% MR2S: Scaling any triangle
% uniformly does not change its type.

%% Parameters (old test)
array[1..3] of int: tuple1;
−2..3: r1;

%% Variables (follow−up test)
array[1..3] of var int: tuple2;
var −2..3: r2;
var int: k;

%% Follow−up constraints
constraint k>1;
constraint tuple2[1] = k∗tuple1[1];
constraint tuple2[2] = k∗tuple1[2];
constraint tuple2[3] = k∗tuple1[3];
constraint r2 = r1 ; % Scaled triangle is of same type.

%% Solution
solve satisfy;
```

Listing 3 Sample MiniZinc model of MR2S (valid on any path)

1. Definition of the initial test case (parameters). The initial test case $t_1 = ((a_1, b_1, c_1), r_1)$, in MiniZinc, is translated into parameters: an array of 3 elements for the side lengths, `array[1..3] of int: tuple1` for the input, and `-2..3: r1` for the output.
2. Definition of the follow-up test case (variables). The follow-up test case $t_2 = ((a_2, b_2, c_2), r_2)$ in MiniZinc, is translated into variables: another array for the side lengths, `array[1..3] of var int: tuple2`, and the output as `var -2..3: r2`.
3. Module of path-specific constraints corresponding to the initial test case (Listings 3 and 5).
4. Set of constraints corresponding to the follow-up test case (Listings 3 and 5).
5. A `solve satisfy;` statement to obtain the solution (Listings 3 and 5).
6. A data file with the extension `dzn`, with the same name as the MiniZinc model (Listing 4).

Listing 3 shows the MiniZinc model for MR2S, in which each element of the initial test case is multiplied by a constant greater than one, and a follow-up test case belonging to the same path as the initial test case is obtained. Listing 4 has an example of a data file that is used for the MR2S parameters.

```
%% Data file of MR2S                                    1
%% Path 5 (scalene)                                     2
tuple1 =[ 4, 3, 2 ];                                    3
r1 = 1;                                                 4
```

Listing 4 Sample MiniZinc data file of MR2S

Listing 5 has the MiniZinc model for MR1D. From an initial test case on any path except path 1, the model produces a follow-up test case on path 1.

Although the definition of MRs is broader as indicated by Chen, Kuo, Tse, and Zhi Quan Zhou (2003), *an expected relation among the inputs and outputs of multiple executions*, the MRs designed guided by METRECoP obey the outline where the initial test case inputs are disturbed to obtain subsequent test cases where the inputs and outputs have changed, as we described in Section 3.4.

Beyond this template used in METRECoP, MiniZinc allows for defining constraints that combine initial and follow-up inputs and outputs in arbitrary ways. This means that it should be possible to define MRs where the follow-up cases can be obtained with other types of disturbances on the outputs, or where multiple initial tests can be combined to produce a follow-up test. In short, we argue that the set of MRs can MiniZinc can express is broader than those which are obtained by the current version of MiniZinc.

As an example, let us consider how to represent in MiniZinc an MR where two initial test cases are combined into a new follow-up test case. Given a first initial test case $t_1 = (I_1, O_1)$, a second initial test case $t_2 = (I_2, O_2)$, and a follow-up test case $t' = (I', O')$, such an MR would follow this general pattern:

MRiC2 (Combine two initial test cases into one follow-up test case):

$$p(t_1, t_2) \wedge f(I_1, I_2, I') \Longrightarrow g(O_1, O_2, O')$$

The predicate p indicates if the MR is applicable for that pair of initial test cases. The function f relates the inputs of the follow-up test to those of the initial tests, and the function g relates the outputs of the follow-up test to those of the initial tests.

For the triangle program, one MR that would follow this pattern (MRiC2) is implemented in MiniZinc in Listing 6). Given two scalene right-angle triangles with side lengths (h_1, j, b_1) and (h_2, j, b_2), where $h1$ and $h2$ are the hypothenuses and $h1 \neq h2$, the two triangles(two initial test cases: t_1 and t_2) can be joined on the catheti of length j, obtaining an scalene triangle (follow-up test case, t_3) with side lengths $(h_1, h_2, b_1 + b_2)$. Please note that due to the use of floating-point arithmetic, the Pythagorean theorem $h^2 = a^2 + b^2$ has been approximated as $h^2 - (a^2 + b^2) < \epsilon$, where ϵ is a very small value (10^{-6} in this case).

This MR can be solved by MiniZinc using two initial scalene triangles $(5, 3, 4)$ and $(6.708, 3, 6)$ as inputs, producing the follow-up scalene triangle

```
%% MR1D: From Paths 2-5 to Path 1:
%% follow-up output must be -1.

%% Parameters (old test)
array[1..3] of int: tuple1;
-2..3: r1;

%% Variables (follow-up test)
array[1..3] of var int: tuple2;
var -2..3: r2;

%% Path-specific constraints
% Check all the sides of triangle > 0
constraint forall(1 in 1..3) (tuple1[1] > 0);
% Check all the sides form a triangle:
constraint tuple1[1] + tuple1[2] > tuple1[3]
/\ tuple1[1] + tuple1[3] > tuple1[2]
/\ tuple1[2] + tuple1[3] > tuple1[1];
% Check old test does not belong to path 1
constraint r1 != -1 ;

%% Follow-up constraints
constraint tuple2[1] = 0 ;
constraint tuple2[2] = tuple1[2];
constraint tuple2[3] = tuple1[3];
constraint r2 = -1 ; % Not a triangle anymore

%% Solution
solve satisfy ;
```

Listing 5 Sample MiniZinc model of MR1D (from any path to path 1)

$(5, 6.708, 10)$. This is one example of an MR which would not be produced by the current version of METRECoP, but which can be expressed in MiniZinc.

4.6 Iterative test generation and MR refinement

The final step of the methodology is to create an extended test suite from an initial test suite, produced by the path constraints of Section 4.3. The steps of Section 3.6 will be followed:

1. The first step is the selection of the technique to evaluate the tests. This will be done using the mutation testing technique, which allows the quality of a set of tests to be evaluated. The technique applies mutation operators that generate versions of the original program (*mutants*) with

```
%% MR1C2: From two right−angle scalene triangles
%% (hyp1, joined, base1) and (hyp2, joined, base2)
%% where hyp1 != hyp2, follow−up triangle
%% (hyp1, hyp2, base1 + base2) must be scalene.

%% Parameters for initial test cases
array[1..3] of float: tuple1; 1..3: r1;
array[1..3] of float: tuple2; 1..3: r2;

%% Variables (follow−up test)
array[1..3] of var float: tuple3; var 1..3: r3;

%%% Initial constraints

% Side lengths must be positive
constraint forall (l in 1..3) (tuple1[l] > 0);
constraint forall (l in 1..3) (tuple2[l] > 0);

% First values are hypothenuse of right−angle triangles
constraint tuple1[1]*tuple1[1]
  − tuple1[2]*tuple1[2] − tuple1[3]*tuple1[3] < 1e−6;
constraint tuple2[1]*tuple2[1]
  − tuple2[2]*tuple2[2] − tuple2[3]*tuple2[3] < 1e−6;

% Second values are catheti to be joined (same length)
constraint tuple1[2] = tuple2[2];

% Hypothenuses are different in length
constraint tuple1[1] != tuple2[1];

%% Follow−up: (hyp1, hyp2, base1 + base2)
constraint tuple3[1] = tuple1[1];
constraint tuple3[2] = tuple2[1];
constraint tuple3[3] = tuple1[3] + tuple2[3];

% Extra check: follow−up sides form a triangle
constraint tuple3[1]+tuple3[2] > tuple3[3] /\
    tuple3[1]+tuple3[3] > tuple3[2] /\
    tuple3[2]+tuple3[3] > tuple3[1];

constraint r3 = 1; % scalene

%% Solution
solve satisfy;
```

Listing 6 Sample MiniZinc model of MR with two initial test cases and one follow-up test case

Table 1 Metamorphic relations for an initial triangle (a, b, c)

Name	Description
MR1Sab	Same path, exchange of two values: $a \leftrightarrow b$
MR1Sac	Same path, exchange of two values: $a \leftrightarrow c$
MR1Sbc	Same path, exchange of two values: $b \leftrightarrow c$
MR2S	Same path, scaling any triangle uniformly does not change its type
MR1D	Changed path, a is set to 0, moving to Path 1 (any of $\{a, b, c\}$ is ≤ 0)
MR2D	Changed path, one of $\{a, b, c\}$ ¿ 0 is copied to the others, moving to Path 3 (equilateral)
MR3Dab	Changed path, a is copied to b, moving to Path 4 (isosceles)
MR3Dac	Changed path, a is copied to c, moving to Path 4 (isosceles)
MR3Dbc	Changed path, b is copied to c, moving to Path 4 (isosceles)
MR3Deq	Changed path, from equilateral (Path 3), incrementing a and b by $k > 0$, moving to Path 4 (isosceles)
MRD4b	Changed path, from any path with all values > 0, replacing b by k where $k > a + c$, moving to Path 2 (all > 0, not a triangle)
MRD4c	Changed path, from any path with all values > 0, replacing c by k where $k > a + b$, moving to Path 2 (all > 0, not a triangle)

small changes. If a test case gives a different result from the original program, the mutant is said to be *killed*: the test case has detected the change in the mutant. Ideally, a high quality test suite should detect all mutants, except those that are semantically equal (*equivalent mutants*). To apply this technique, we use the MuBPEL tool (García-Domínguez, Estero-Botaro, & Medina-Bulo, 2012), which implements the mutation operators of Estero-Botaro, Boubeta-Puig, Liñeiro-Barea, and Medina-Bulo (2012); Estero-Botaro, Palomo-Lozano, and Medina-Bulo (2010) for the WS-BPEL web service orchestration language.

2. Initial test cases generated with MiniZinc (Section 4.3) are available for each path in the program. MuBPEL is used with these test cases and the WS-BPEL composition of the triangle which follows the flowchart in Figure 2. The number of mutants killed by this initial test suite is noted.
3. Two rounds of test case generation are performed from the 10 MRs initially designed in Section 4.5 (the first group in Table 1). At the end of each round, MuBPEL is used with these test cases, and it is checked if they can kill mutants that the initial set was unable to kill.
4. Analysing the surviving mutants after the second round, it was observed that repeating the same metamorphic relations would not be useful. Instead of repeating the process once more, 2 new MRs were added (second group in Table 1), producing new test cases that killed the non-equivalent surviving mutants.

4.6.1 Execution of initial test suite

The initial test suite is the one obtained in Section 4.3 through MiniZinc:

1. Path 1: ((-10000000,1,1), -1).
2. Path 2: ((2,1,1), -2).
3. Path 3: ((1,1,1), 3).
4. Path 4: ((2,2,1), 2).
5. Path 5: ((4,3,2), 1).

By using the MuBPEL tool, we obtain with the Triangle composition that 33 mutation operators can be applied to the original composition in order to obtain mutated versions of it. We apply them to the composition and derive 235 mutants, which are versions of the original composition in which a part has been modified according to the mutation operators mentioned above. Running the composition against the initial test suite, we get that **163** mutants are killed, so 72 remain alive. This is more than 69% of the generated mutants.

4.6.2 First round of follow-up test cases

Following this, we try to apply the MRs previously specified and implemented and described in the first group of Table 1 in MiniZinc to obtain sets of follow-up test cases, and check if those sets are able to kill some of the mutants that are left alive and that the initial set was unable to kill.

In order to explain the process followed with all MRs, the steps followed and the results obtained will be detailed for some MRs:

1. MR1Sab (Same path, exchange of two values: $a \leftrightarrow b$): Applying this MR to the initial 5 test cases produced the following set of test cases: ((1, -10000000, 1),-1), ((1, 2, 1), -2), ((1, 1, 1), 3), ((2, 2, 1), 2), ((3, 4, 2),1). The composition is then executed with this reduced set of test cases (removing the tests that have already been run), and MuBPEL is used with the mutants that remain alive. In this case, the test case set obtained has managed to kill **22** mutants, out of the 72 that were left alive.
2. MR1D (Changed path, a is set to 0, moving to Path 1 (any of $\{a, b, c\}$ is ≤ 0), Listing 5): Applying this MR to 4 of the 5 initial test cases (except for the test case already on path 1), we get the following set of test cases: ((0,1,1), -1),((0,1,1), -1),((0,2,1), -1), ((0,3,2), -1). Following this, the composition is executed with these test cases and the mutants left alive, using MuBPEL. In this case, the set of test cases obtained has managed to kill *3* new mutants, among the 72 that were left alive.

As a final result of this round, a total of 37 more mutants are killed, out of the 72 survivors of the initial set. This represents more than 50% of these mutants. Therefore, 35 mutants remain alive.

Furthermore, of the 10 implemented MRs, only 5 MRs have obtained test cases that have killed mutants (MR1Sab, MR1Sac, MR1Sbc, MR1D and MR3Dac). There are 5 MRs (MR2S, MR2D, MR3Dab, MR3Dbc and MR3Deq)

whose follow-up test cases have not killed any mutants, however they have generated new test cases that can be used in the next iteration as initial cases for other MRs.

4.6.3 Second round of follow-up test cases

Once testing is done with the initial test suite for the implemented MRs, the aim is to further evaluate the MRs, taking as initial test suite all the follow-up test cases obtained that are applicable to each of the MRs.

In this second round, a record is made of the steps taken and the results obtained for each MR. In this case we start from all the test cases obtained in the first iteration. We remove duplicate test cases, and test cases that already appeared in the initial test suite. This leaves 21 different test cases to be used as the initial test cases in the second iteration.

The 10 existing MRs were applied to these 21 test cases. The steps for some of the MRs and the overall results are discussed below:

1. MR1Sab (Same path, exchange of two values: $a \leftrightarrow b$). this MR produced 20 valid test cases. Of these, there are only 11 new test cases to be tested with the composition and the mutants, using MuBPEL.

   Next, the 35 mutants left alive from the previous iteration were run against these 11 test cases. In this case, **3** mutants die through the same test case: ((1, 0, 1), -1). We note that this test case comes from the test case ((0, 1, 1), -1), generated with the MR1D, in the first iteration. Therefore, 32 of the 35 mutants in this second round are still alive.
2. MR1D (Changed path, a is set to 0, moving to path 1 (any of $\{a, b, c\}$ is $\leq$ 0), Listing 5): Applying MR1D to the 21 test cases (except those already on path 1), 3 follow-up test cases are obtained. These are: {((0, 3, 4), -1), ((0, 4, 4), -1), ((0, 3, 3), -1)}.

   Next, the 35 mutants left alive from the previous iteration were run against these 3 test cases. In this case, no new mutants were killed, so this next set of test cases obtained with MR1D did not serve to improve coverage in this second round.

To summarize, a total of 7 mutants have been killed in this second iteration, which is about 20% of the mutants left alive. Three mutants have been successfully killed with test cases obtained from MR1Sab, and the remaining four with 2 test cases obtained from MR1Sac: the case ((1, 1, 0), -1) kills 3 mutants and the case ((3, 3, 4), 2) kills the fourth mutant.

Note that the following test case ((3, 3, 4), 2) comes from the initial case ((4, 3, 3), 2), obtained in the first iteration by MR3Dbc, and that in that iteration, its follow-up test cases did not succeed in killing any mutants. However, when combined in this second iteration with MR1Sac, a test case is obtained that does kill a mutant. This suggests that there are MRs that, although they do not kill mutants themselves, may be necessary for other MRs to generate test cases that detect errors in the code in future iterations.

Furthermore, in this second stage, as there are fewer mutants left alive, it is more difficult to find test cases that kill these remaining mutants. It may be that some of these mutants are equivalent or difficult to kill, or that there are no MRs left to generate cases that will kill them. Therefore, an analysis of the results and of the mutants left alive is necessary to conclude whether all mutants are equivalent, whether a new iteration is necessary, or whether new MRs that kill some mutants need to be specified and implemented.

4.6.4 Refinement of MRs through mutant analysis

Up to this point, 44 mutants (37 in the first iteration and 7 in the second iteration) of the initial 72 mutants have been killed, i.e. 61.11%. This leaves 28 mutants alive, which are analyzed below:

- The 19 mutants for the ECC operator [5] from (Estero-Botaro et al., 2010), the 5 mutants for the EAP operator[6], and the 2 mutants for the EAN operator[7] are equivalent (as defined in UCASE Research Group (2020)).
- The remaining 2 mutants correspond to the ERR operator [8], (according to Estero-Botaro et al. (2010)). In this case they are not equivalent mutants, so it is possible to find test cases that kill them. Checking the test cases initially generated by MiniZinc, it was identified that by making changes to the initial test case ((1, 1, 1), 3) these mutants could be killed. The first mutant could be killed by substituting the second side with a value of $k > a + c$, obtaining ((1, 3, 1), -2). The second mutant could be killed by substituting the third side with a value $k > a + b$, obtaining ((1, 1, 3), -2).

Therefore, MRs were designed to automatically make this type of change to the initial test suite. The two new MRs are called MR4Db and MR4Dc, which generate test cases that kill both mutants and are described below, incorporating them into the set of 10 initial MRs. They constitute the second group of MRs shown in Table 1:

- MR4Db: From any path with all sides greater than 0, a test case is obtained in which the sides do not form a triangle by substituting the b side with a value of $k > a+c$, i.e. whose value is greater than the sum of the other sides.
- MR4Dc: From any path with all sides greater than 0, a test case is obtained in which the sides do not form a triangle by substituting the c side with a value of $k > a+b$, i.e. whose value is greater than the sum of the other sides.

After adding these MRs, a total of 46 of the 72 mutants that survived the initial set have been killed, and the remaining 26 have been found to be equivalent. With these 12 MRs, it has been possible to generate tests cases that kill all non-equivalent mutants. The results per phase and MR can be seen in Table 2.

[5] Replaces a path operator (/, //) by another of the same type
[6] Inserts the positive absolute value in an arithmetic expression
[7] Inserts the negative absolute value in an arithmetic expression
[8] replaces a relational operator ($<, \leq, >, \geq, =, \neq$) with another of the same type

Table 2 Test cases generated and mutans killed by test source in order of generation. The final 26 remaining mutants were found to be equivalent.

Phase	Source	Generated tests	New tests	Killed	First-time killed	Remaining
Initial	Path constraints	5	-	163	-	72
Round 1	MR1Sab	5	3	22	22	50
	MR1Sac	5	4	26	10	40
	MR1Sbc	5	2	4	2	38
	MR2S	5	5	0	0	38
	MR1D	4	3	3	3	35
	MR2D	4	1	0	0	35
	MR3Dab	2	1	0	0	35
	MR3Dac	2	1	3	0	35
	MR3Dbc	1	1	0	0	35
	MR3Deq	1	0	0	0	35
Round 2	MR1Sab	20	11	3	3	32
	MR1Sac	20	8	4	4	28
	MR1Sbc	20	5	0	0	28
	MR2S	20	10	0	0	28
	MR1D	14	3	0	0	28
	MR2D	18	1	0	0	28
	MR3Dab	4	1	0	0	28
	MR3Dac	5	1	0	0	28
	MR3Dbc	4	1	0	0	28
	MR3Deq	2	2	0	0	28
Refinement	MR4Db	4	3	1	1	27
	MR4Dc	4	3	1	1	26

Consequently, this methodology requires iterations of both using and designing MRs until a satisfactory result is achieved: in terms of a set of test cases that are sufficient to detect errors, and in terms of MRs that are capable of generating test cases that detect those errors.

5 Discussion

Based on the previous results, we can answer the research questions as follows:

1. **RQ1**: *Can a set of metamorphic relations be identified from a selection of execution paths across the source code of a system?*

   We obtained a first set of MRs taking into account whether perturbations in the values of the inputs of the initial test cases produced follow-up test cases that maintained the execution path or changed to another one. These MRs have been grouped using the suffixes "S" (same path) and "D" (different path), e.g. MR2S, MR1D, etc. This first set consists of 10 MRs (4 S and 6 D), considering 5 different paths in the execution of the software according to the flowchart in Fig. 2.

Nevertheless, the above 10 MRs did not kill all remaining mutants: 28 survived and were manually inspected. Two mutants were found to be non-equivalent, meaning that additional MRs were required. For this reason, METRECoP includes an MR refinement stage, in which other software testing techniques can be integrated as sources of additional information. In this work, mutation analysis has been integrated into this final step of the methodology to find new MRs that had not yet been found. METRECoP combined the first MRs and the two new MRs into an extended set, allowing their interaction to produce a larger set of test cases over a few iterations.

Overall, the case study has shown that studying execution paths can produce a set of MRs covering a relevant subset of the program logic, but this set may need to be refined through manual inspection of surviving mutants.

2. **RQ2**: *Can existing constraint programming systems express the extracted metamorphic relations?* All the MRs in the case study were mapped to MiniZinc, with no need for custom extensions or modifications, following a step-by-step approach (Section 4.5). This process uses parameters for the initial test (set by the user) and variables for the follow-up test (computed by the solver), and collects constraints from several sources: the program itself, the original path, the target path, and the intended perturbation. Being a declarative language, MiniZinc only needs to be told about which constraints to solve, meaning that the mapping can be kept more concise and easier to read, as it does not need to specify a solution strategy. Some of these constraints (e.g. program constraints) are shared between multiple MRs, so they only need to be written once and then they can be reused (MiniZinc itself includes some capabilities for modularity, with its `includes` keyword).

   In addition, Section 4.5 shows that MiniZinc can also express MRs that do not follow the "perturbation to stay on same path or change path" approach followed in METRECoP: constraint solvers can implement MRs beyond those of the proposed methodology.

   In general, the experience with MiniZinc has been a positive one for this case study, which is aimed at a simple triangle classification problem. For more complex pieces of software, the test cases may require more complex or different data types. This may require using or even creating further extensions to MiniZinc: for instance, in Amadini et al. (2016) is proposed an extension to allow MiniZinc to handle strings.

3. **RQ3**: *How effective are the test cases generated from the metamorphic relations at detecting defects?*

   The initial tests killed 163 out of the original set of 235 mutants, leaving 72 mutants alive. The first set of 10 MRs was applied twice: in "Round 1" of Table 2, 21 tests (28% of the full set) were generated from the 5 initial tests, killing 37 (51.39%) of the 72 remaining mutants. Round 2 simply reran the MRs on the 21 tests from Round 1, to allow the MRs to interact with each other: this resulted in 43 new tests (57.3% of the full set), which killed 7 more mutants.

The results from Round 2 suggests that allowing the same MRs to be applied over multiple rounds can result in test suites with better mutation scores. It should be noted that while only 5 of these 10 MRs generated test cases that detected errors, this does not necessarily mean that they do not provide value: they may still be useful for their interactions with other MRs. For instance, while the tests in Round 1 from MR3Dbc did not kill any mutants by itself, the follow-up tests produced by applying MR1Sac on one of those tests from MR3Dbc did kill a mutant in Round 2.

The original MRs by themselves did not kill the last two non-equivalent mutants: two new MRs had to be designed in the refinement step (Section 4.6.4), producing 6 new tests that killed them. This suggests that while the initial MRs produced from studying the program flowchart can produce good mutation scores, reaching the highest values may still require manual inspection of the surviving mutants. One advantage of using MRs is that as soon as a non-equivalent mutant is found, it is possible to create an MR for it and then use it on all the tests so far, and see if the resulting set may kill other mutants beyond this one.

6 Threats to validity

The results in Section 4.6 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. However, the results are subject to some 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).

6.1 Internal threats

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

Twelve MRs were defined by only modifying one of the initial test values in each of them. This has 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 if 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 from the program: the MRs operate on the assumption that the initial test is valid. Applying an MR as implemented above to an invalid test case will only produce another invalid test case.

6.2 External threats

The chosen case study (the Triangle composition) is a small one, in order to facilitate the understanding of the process. However, the proposed approach of using constraint solvers to formalize MRs should be extensible to larger programs written in other languages — particularly, the MiniZinc solver has been used for much larger optimisation problems. Most limitations would come from the expressive power of the solver.

There may be test cases or MRs that cannot be represented with the language MiniZinc as is. For instance, we have not evaluated 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 character strings (Amadini et al., 2016; Caballero, Stuckey, & Tenorio-Fornés, 2015). These extensions must be supported by the solver, though: MiniZinc is just a common constraint programming language.

In this case study, all 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.

7 Related works

In this section, publications related to the methodology proposed in this work and to the formalisms and techniques used are described.

7.1 Formalization of Metamorphic Relations

As mentioned, there are some studies that guide or propose some formats for MRs. These studies will be briefly described below. In Zhang et al. (2014), Zhang et al. proposed a search-based approach to automatic inference of polynomial MRs for a program under test. In particular, they use a set of parameters to represent a particular class of MRs, which they refer to as polynomial MRs, and turn the problem of inferring MRs into a problem of searching for suitable values of the parameters. the tool implemented is MRI (MR inferrerer). In this proposal the MRs must obey a certain scheme. Furthermore, they are only suitable for numerical values. On the other hand, 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 type MRs, but all those MRs that can be represented through the constraint programming language (e.g. existential/universal quantification or arrays).

In another paper Sun et al. (2016), propose 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. (2017), 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 formal enough to be processed by a computer and checked for consistency.

In another more recent work Segura, Durán, Troya, and Ruiz-Cortés (2019), several ways to represent of MRs are proposed, but for a specific domain: query languages such as SQL. Therefore, the proposal is not sufficiently homogeneous to be applied to different domains, as proposed in this paper.

7.2 Identification of Metamorphic Relations

Sun et al. presented a framework to support metamorphic testing of web services in Sun et al. (2011). The framework included components to generate tests given a set of MR descriptions, execute them and evaluate their results, but did not have explicit guidance on how to identify the MRs, or how to formalise their descriptions into a machine-readable format as in the present work.

H.Liu, X.Liu, and T.Y.Chen (2012), propose the construction of composite MRs by treating the follow-up test of an MR as the start test of another MR: the initial MRs still need to be manually identified. In our approach, while we do provide support for identifying the initial set of MRs, we follow a similar strategy where the second round of follow-up test cases is produced by re-applying the MRs to the first-round of test cases. We have also noticed that some of these follow-up test cases can kill mutants that the individual MRs would not be able to kill. In a way, this confirms the positive results from Liu, while remaining simpler to use than having to explicitly identify composite MRs.

Chen et al. (2016), 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 if the pair is usable for producing an MR. Test frames are produced by partitioning the input domain, and the approach is supported by a tool called MR-GEN. In a way, this can be considered as a black-box approach to designing MRs, where only the inputs and outputs are relevant. METRECoP follows a white-box approach which considers the available paths through the program: the paths themselves may produce a certain partition of the input domain, and the follow-up test may involve changing to a different path (and therefore a different part of that input domain). Unlike METRIC, which can only ask the user to decide if a particular arbitrary pairing of test frames is usable, the use of paths in our methodology provides additional guidance on the constraints to impose on the follow-up test case.

Sun et al. have recently proposed an improvement of METRIC called METRIC+ (Sun et al., 2019), which also performs partitioning on the output

domain. In our case study, none of our MRs have required placing conditions on the outputs of the original test case: the MRs have mostly focused on specifying the expected output given a change in the inputs, sometimes motivating the switch to a different execution path. We envision that the idea of partitioning the output could be adapted to our approach by adding conditions on the original output. In any case, METRIC+ can be still considered as a black-box approach that ignores the logic of the program, whereas METRECoP is a white-box approach.

To summarize, these works either only provide partial guidance on how to identify the MRs and/or generate tests from them (H.Liu et al., 2012; Sun et al., 2011), or follow a black-box approach that uses input/output domain partitioning ignoring the program logic (Chen et al., 2016; Sun et al., 2019). METRECoP guides the entire process, from the identification of the initial MRs, to the generation of multiple rounds of follow-up test cases, while following a white-box approach that takes into account the execution paths of the program. METRECoP formalises the MRs in a constraint programming language for which mature solvers exist.

7.3 Constraint solving and MT

Constraint solving and metamorphic testing have been combined in different ways in the past. However, the existing works have focused on different aspects, and their approaches are unlike ours. Firstly, Gotlieb and Botella (2003) use constraint logic programming to find test cases that do not satisfy a given MR. They present some 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 (Axlog Ingenierie, 2002).

On the other hand, a recently published paper of Akgün, Gent, Jefferson, Miguel, and Nightingale (2018), propose using MT to validate constraint programming systems. The complexity of these systems makes testing them difficult otherwise. Specifically, they evaluate using MT to validate the Minion CPS (Gent et al., 2006). They show that MT is effective in finding artificial bugs introduced through random code mutation. Clearly, the meaning of this paper is very different from the proposal of our work.

8 Conclusion and future work

The methodical identification and formalisation of MRs has been highlighted as a challenge in the MT literature. Existing methodologies use natural language to describe the identified MRs, use ad-hoc test generation algorithms, or do not use the internal logic of the program to guide their design. In this paper, we have proposed METRECoP, an end-to-end methodology to apply Metamorphic Testing that uses the internal logic of the program to guide the design of MRs, and formalizes MRs through an existing constraint programming languages, specifically MiniZinc. The use of MiniZinc allows for

leveraging mature constraint solvers, rather than ad-hoc generation algorithms as in other methodologies.

The set of initial test cases is designed and implemented through 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 that a) stay in the same path but change the output significantly, or b) change to a different execution path. These MRs are formalized as well in MiniZinc, so the solver can automatically produce the follow-up tests.

The final step in METRECoP is an iterative test generation and MR refinement process, which allows for the integration of an arbitrary test sufficiency criterion. For the case study in this paper (a WS-BPEL version of the Siemens triangle program), we have chosen mutation analysis as the criterion, using the MuBPEL tool. This process consists of successive *rounds* of follow-up test cases, terminating when the number of killed mutants converges. In each round, the previous tests (starting from the initial tests) are given to all MRs, producing a new set of follow-up tests. We have obtained promising results from this, killing all the 46 non-equivalent mutants of the program under test in 2 rounds while motivating the design of 2 additional MRs. Unlike IMT, METRECoP does not require manually designing MR sequences.

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 give the designer strong evidence that the candidate MR may be a logical contradiction and may be thus rejected. We plan to study the integration of these additional applications to an extended version of METRECoP in the future.

We are considering applying our insights from METRECoP to other MT methodologies that only consider inputs and outputs, following a black-box approach. We consider that these methodologies could benefit from an iterative approach for refining both tests and MRs that does not require manual design of MR sequences, and from the use of constraint programming languages.

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

Declarations

Funding statement. This work was partly supported by the Spanish Ministry of Science and Innovation and the European Regional Development Fund (ERDF) under project FAME (RTI2018-093608-B-C33).

Declaration of competing interest. The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Authors' contribution statement. All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by M. Carmen de Castro-Cabrera, Antonio García-Dominguez and Inmaculada Medina-Bulo. The first draft of the manuscript was written by M. Carmen de Castro-Cabrera and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Data Availability statement. The datasets generated and analysed during the current study are available from the corresponding author upon request.

References

Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P. (2018). Metamorphic testing of constraint solvers. J. Hooker (Ed.), *International conference on principles and practice of constraint programming* (p. 727-736). Springer.

Amadini, R., Flener, P., Pearson, J., Scott, J.D., Stuckey, P.J., Tack, G. (2016). MiniZinc with Strings. *CoRR*, *abs/1608.03650*. Retrieved from http://arxiv.org/abs/1608.03650 arXiv:1608.03650

Ammann, P., & Offutt, J. (2016). *Introduction to software testing* (2nd ed.). New York: Cambridge University Press.

Anand, S., Burke, E.K., Chen, T.Y., et al. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, *86*(8). DOI: 10.1016/j.jss.2013.02.061.

A. Oplobedu, J.M., & Tourbier, Y. (1989). Charme: Un langage industriel de programmation par contraintes, illustré par une application chez Renault. . EC2 (Ed.), *Proceedings of the 9th international workshop on expert systems and their applications* (Vol. 1, pp. 55–70). Springer.

Apt, K. (2003). *Principles of constraint programming.* Cambridge University Press.

Axlog Ingenierie (2002). *Inka–v1 user's manual* (Tech. Rep.). Thales Airborne Systems.

Beizer, B. (1990). *Software testing techniques (2nd ed.).* New York, NY, USA: Van Nostrand Reinhold Co.

Božić, J. (2022). Ontology-based metamorphic testing for chatbots. *Software Qual J*, *30*(1), 227–251. DOI: https://doi.org/10.1007/s11219-020-09544-9.

Caballero, R., Stuckey, P.J., Tenorio-Fornés, A. (2015). Two type extensions for the constraint modeling language MiniZinc. *Science of Computer Programming*, *111*, 156 - 189. DOI: 10.1016/j.scico.2015.04.007.

Chen, T.Y. (1998). Metamorphic testing: A new approach for generating next test cases. *HKUSTCS98-01*.

Chen, T.Y. (2010). Metamorphic testing: A simple approach to alleviate the oracle problem. L. O'Conner (Ed.), *Proceedings of the 5th ieee international symposium on service oriented system engineering.* IEEE Computer Society.

Chen, T.Y., Kuo, F., Tse, T.H., Zhi Quan Zhou. (2003, Sep.). Metamorphic testing and beyond. L. O'Brien, N. Gold, & K. Kontogiannis (Eds.), *Eleventh annual international workshop on software technology and engineering practice (2003)* (p. 94-100). 10.1109/STEP.2003.18

Chen, T.Y., Kuo, F.-C., Liu, H., Poon, P.-L., Towey, D., Tse, T.H., Zhou, Z.Q. (2018, January). Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, *51*(1), 4:1–4:27. DOI: 10.1145/3143561.

Chen, T.Y., Poon, P.-L., Xie, X. (2016). Metric: METamorphic relation identification based on the category-choice framework. *Journal of Systems and Software*, *116*, 177–190. DOI: 10.1016/j.jss.2015.07.037.

Data61 research network, CSIRO (2015). *MiniZinc.* https://www.minizinc.org/index.html. (Date of last access: 4 March 2022)

de Castro-Cabrera, M.C., García-Domínguez, A., Medina-Bulo, I. (2019, May). Using constraint solvers to support metamorphic testing. X. Xie, P. Poon, & L.L. Pullum (Eds.), *Proceedings of the 4th international workshop on metamorphic testing* (pp. 32–39). IEEE. 10.1109/MET.2019.00013

de Castro-Cabrera, M.d.C., García-Dominguez, A., Medina-Bulo, I. (2020). Trends in prioritization of test cases: 2017-2019. T. Hung Chih-Cheng; Cerny (Ed.), *Proceedings of the 35th annual acm symposium on*

applied computing (p. 2005–2011). New York, NY, USA: Association for Computing Machinery. 10.1145/3341105.3374036

Estero-Botaro, A., Boubeta-Puig, J., Liñeiro-Barea, V., Medina-Bulo, I. (2012, 01). Operadores de mutación de Cobertura para WS-BPEL 2.0. L.E. Ruíz-Cortés A. e Iribarne (Ed.), *Jisbd 2012: Actas de las xvii jornadas de ingeniería del software y bases de datos.*

Estero-Botaro, A., Palomo-Lozano, F., Medina-Bulo, I. (2010). Quantitative Evaluation of Mutation Operators for WS-BPEL Compositions. R. Bilof (Ed.), *Proceedings of III International Conference on Software Testing, Verification, and Validation Workshops* (pp. 142–150). Paris, France: IEEE Computer Society. 10.1109/ICSTW.2010.36

Fraser, G., & Arcuri, A. (2014, December). A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.*, *24*(2). DOI: 10.1145/2685612.

García-Domínguez, A., Estero-Botaro, A., Medina-Bulo, I. (2012). MuBPEL: una herramienta de mutacicón firme para WS-BPEL 2.0. C. Calero-Muñoz & Ángeles Saavedra-Places (Eds.), *XVI JISBD.*

Gent, I.P., Jefferson, C., Miguel, I. (2006). Minion: A fast scalable constraint solver. G.B. et al. (Ed.), *Ecai* (Vol. 141, pp. 98–102). IOS Press.

Gotlieb, A., & Botella, B. (2003). Automated metamorphic testing. F.M. Titsworth (Ed.), *Proceedings of compsac 2003* (pp. 34–40). DOI: 10.1109/CMPSAC.2003.1245319.

H.Liu, X.Liu, T.Y.Chen. (2012). A new method for constructing metamorphic relations. R. Bilof (Ed.), *2012 12th international conference on quality software* (pp. 59–68).

Hui, Z.-W., Huang, S., Chua, C., Chen, T.Y. (2019). Semiautomated metamorphic testing approach for geographic information systems: An empirical study. *IEEE Transactions on Reliability*, *69*(2), 657–673.

Jaffar, J., & Maher, M.J. (1994). Constraint logic programming: A survey. *The journal of logic programming*, *19*, 503–581.

Johnson, T.E. (1963). Sketchpad III: a computer program for drawing in three dimensions. C.-H. Press (Ed.), *Proceedings of the may 21-23, 1963, spring joint computer conference* (pp. 347–353).

Jussien, N., Rochart, G., Lorca, X. (2008, May). Choco: an open source Java constraint programming library. L.P.A. Trick (Ed.), *CPAIOR'08 workshop on open-source software for integer and contraint programming (ossicp'08).* Paris, France: Springer.

Kjellerstrand, H. (2012). *Comparison of CP systems — and counting.* Available from http://www.it.uu.se/research/group/astra/SweConsNet12/hakan.pdf. (Date of last access: 4 March 2022)

Kuchcinski, K., & Szymanek, R. (2013). *JaCoP — Java Constraint Programming Solver.* Lund University Publications.

Lauriere, J.-L. (1978). A language and a program for stating and solving combinatorial problems. *Artificial intelligence*, *10*(1), 29–127.

Mcgraw, G., Michael, C., Schatz, M. (1997). Generating software test data by evolution. *IEEE Transactions on Software Engineering*, *27*, 1085–1110.

Morán, J., Bertolino, A., de la Riva, C., Tuya, J. (2018, Sep.). Automatic testing of design faults in mapreduce applications. *IEEE Transactions on Reliability*, *67*(3), 717-732. DOI: 10.1109/TR.2018.2802047.

Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G. (2007). MiniZinc: Towards a standard CP modelling language. C. Bessière (Ed.), *Principles and practice of constraint programming – cp 2007* (pp. 529–543). Berlin, Heidelberg: Springer Berlin Heidelberg.

Núñez, A., Cañizares, P.C., Núñez, M., Hierons, R.M. (2020). Tea-cloud: A formal framework for testing cloud computing systems. *IEEE Transactions on Reliability*.

O'Conner, L. (2021). *Proceedings of 6th IEEE/ACM International Workshop on Metamorphic Testing.* Virtual event: IEEE. 10.1109/MET52542.2021

Puget, J. (1994). *A C++ implantation of CLP. Ilog Solver collected papers* (Tech. Rep.). ILOG technical report.

Rafeh, R. (2008). The Design of the Zinc Modelling Language. *Constraints*, *13*(3), 229–267.

Rossi, F., Van Beek, P., Walsh, T. (2006). *Handbook of constraint programming*. Elsevier.

Schulte, C., Tack, G., Lagerkvist, M.Z. (2010). Modeling and programming with gecode. *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael*, *1*.

Segura, S., Duran, A., Troya, J., Cortes, A.R. (2017, May). A template-based approach to describing metamorphic relations. L.L. Pullum, D. Towey, U. Kanewala, C. Sun, & M.E. Delamaro (Eds.), *2017 ieee/acm 2nd international workshop on metamorphic testing (met)* (Vol. 00, p. 3-9). Retrieved from doi.ieeecomputersociety.org/10.1109/MET.2017.3 10.1109/MET.2017.3

Segura, S., Durán, A., Troya, J., Ruiz-Cortés, A. (2019). Metamorphic relation patterns for query-based systems. R.S. Bilof (Ed.), *Proceedings of the 4th international workshop on metamorphic testing* (p. 24-31). IEEE Press. 10.1109/MET.2019.00012

Segura, S., Fraser, G., Sanchez, A., Ruiz-Cortes, A. (2016). A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, *PP*(99). DOI: 10.1109/TSE.2016.2532875.

Simonis, H. (1995). The CHIP system and its applications. U. Montanari & F. Rossi (Eds.), *International conference on principles and practice of constraint programming* (pp. 643–646). Springer.

Sun, C., Fu, A., Poon, P., Xie, X., Liu, H., Chen, T.Y. (2019). Metric+: A metamorphic relation identification technique based on input plus output domains. *IEEE Transactions on Software Engineering*. DOI: 10.1109/TSE.2019.2934848.

Sun, C., Wang, G., Mu, B., Liu, H., Wang, Z., Chen, T.Y. (2011, July). Metamorphic testing for web services: Framework and a case study. L.M. Ian Foster & J. Zhang (Eds.), *2011 ieee international conference on web services* (p. 283-290). 10.1109/ICWS.2011.65

Sun, C.-A., Wang, G., Wen, Q., Towey, D., Chen, T. (2016). Mt4ws: An automated metamorphic testing system for web services. *International Journal of High Performance Computing and Networking*, *9*(1-2), 104-115. DOI: 10.1504/IJHPCN.2016.074663.

UCASE Research Group (2020). *Mutation Operators (WS-BPEL).* https://gitlab.com/ucase/public/ws-bpel-testing-tools/-/wikis/MuBPEL#mutation-operators. (Date of last access: 4 March 2022)

University, M., & Data61, C. (2020). *MiniZinc resources.* https://www.minizinc.org/resources.html. (Date of last access: 4 March 2022)

Utting, M., Pretschner, A., Legeard, B. (2012). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, *22*(5), 297-312. DOI: 10.1002/stvr.456.

Valle-Gómez, K.J. (2017). *Triangle WS-BPEL composition.* https://gitlab.com/ucase/public/pfc-kevin-composiciones/-/tree/master/TriangleTFM/bpelContent. (Date of last access: 04 March 2022)

Van Hentenryck, P., & Michel, L. (2005). *Constraint-based local search.* MIT Press.

Weyuker, E.J. (1980). *The oracle assumption of program testing.* Western Periodicals. (13th International Conference on System Sciences)

Wu, P. (2005, July). Iterative metamorphic testing. R. Bilof (Ed.), *29th annual international computer software and applications conference (compsac'05)* (Vol. 2, p. 19-24 Vol. 2). 10.1109/COMPSAC.2005.166

Xie, X., Poon, P.L., & Pullum, L.L. (Eds.). (2019). *Met '19: Proceedings of the 4th international workshop on metamorphic testing.* Montreal, Quebec, Canada: IEEE Press. (ISBN 978-1-7281-2235-9)

Xie, X., Pullum, L.L., & Poon, P.L. (Eds.). (2018). *3rd ACM/IEEE international workshop on metamorphic testing (MET 2018).* Gothenburg, Sweden: ACM. (ISBN 978-1-4503-5729-6)

Yang, G., Khurshid, S., Person, S., Rungta, N. (2014). Property differencing for incremental checking. P. Jalote, L. Briand, & A. van der Hoek (Eds.), *Proceedings of the 36th international conference on software engineering* (pp. 1059–1070).

Zhang, J., Chen, J., Hao, D., Xiong, Y., Xie, B., Zhang, L., Mei, H. (2014). Search-based inference of polynomial metamorphic relations. H. Pei-Breivold (Ed.), *Proceedings of the 29th acm/ieee international conference on automated software engineering* (p. 701–712). New York, NY, USA: Association for Computing Machinery. 10.1145/2642937.2642994

Biographies

M. del Carmen de Castro-Cabrera received the M.S. degree in computer science from the University of Málaga, Spain. She has been with the Department of Computer Science and Engineering, University of Cádiz, Spain, since 1995. She has published some peer-reviewed articles and participated in conference and workshops organization. She is member of the UCASE Software Engineering Research Group, of the Spanish Research Network in Search-Based Software Engineering and of other Research Project in Software Engineering. Her current research interests include software testing and search-based software engineering.

Dr. Antonio Garcia-Dominguez is a Lecturer in computer science at Aston University (United Kingdom). His main research interests are software testing and model-driven engineering: in both of these fields, the increase in system sizes has required the adoption of AI-based approaches and non-relational database technologies to scale up. In addition to over 10 papers in peer-reviewed journals and over 40 papers in conferences and workshops, Antonio is a core contributor in several related open source projects. Some of these projects include the Eclipse Epsilon model management languages and tools, the MuBPEL mutation testing framework for WS-BPEL, or the Eclipse Hawk model indexing framework.

Inmaculada Medina-Bulo received the Ph.D. degree in computer science from the University of Seville, Spain. She has been with the Department of Computer Science and Engineering, University of Cádiz, Spain, since 1995. She has published numerous peer-reviewed articles, participated in conference and workshops organization, and acted as a reviewer for several journals. She is the Main Researcher of the UCASE Software Engineering Research Group, of the Spanish Research Network in Search-Based Software Engineering and of other Research Project in Software Engineering. Her current research interests include software testing, search-based software engineering, the IoT, CEP, and SOA 2.0.