Using constraint solvers to support metamorphic testing

M Carmen de Castro-Cabrera
Department of Computer Science
Universidad de Cádiz
Puerto Real, Cádiz, Spain
maricarmen.decastro@uca.es

Antonio García-Domínguez
SARI, EAS
Aston University
Birmingham, United Kingdom
a.garcia-dominguez@aston.ac.uk

Inmaculada Medina-Bulo
Department of Computer Science
Universidad de Cádiz
Puerto Real, Cádiz, Spain
inmaculada.medina@uca.es

***Abstract*—One of the current challenges in the context of Metamorphic Testing (MT) is the formalization and validation of metamorphic relations (MRs), as there is no single method or homogeneous way of doing it. It is a part of this software testing technique that, unlike others, is not yet developed. On one hand, the fact of having an artifact that formally validates these main elements in MT, facilitates the task for developers and testers and ensures that the technique applied fulfills its function with guarantees. On the other hand, nowadays, there are numerous accessible tools based on highly consolidated and mature constraint solvers that can help in this process of validation. Interpreting MRs as a set of constraints, their validation with these tools is directly applicable. This paper presents a proposal based on a use case, in which MRs are implemented as a set of restrictions. The experiments and the results are described and future lines of research are outlined.**

***Index Terms*—Metamorphic relations, constraint solvers, constraint programming systems, validation**

I. Introduction

In the context of software testing, Metamorphic Testing (MT) has proven to be a useful and effective technique in various fields and disciplines. With a maturity of twenty years [1], and several recent major reports backing it [2], [3], the number of publications per year is increasing and an exclusive event has already been created to discuss and exchange advances in this technique [4], [5], [6]. Metamorphic Testing tries to solve the oracle problem by using Metamorphic Relations (MRs), which are known relationships between inputs and outputs for multiple executions of the program to test [3]. However, there are still challenges to be addressed and aspects that need to be researched and developed, as highlighted in the recent review by Chen et al. [3]. Among them, the second challenge consists of "Systematic MR identification and selection": in the applications carried out, there is neither a unique format nor a systematic way of identifying the MRs. In addition, there are some studies that guide or propose some formats [7], [8], [9], but this aspect of the technique is still in a preliminary stage.

This paper presents a proof of concept with a case study in which several MRs have been represented as a set of constraints. To this effect, we have investigated the most commonly used constraint solvers and have opted for one of them, MiniZinc. With this approach, once the MRs have been represented as constraints, they can be automatically checked for contradictions and used to generate the following test cases.

This paper is structured as follows. Section II highlights the background behind this work. Next, Section III defines the MiniZinc language. Then, Section IV shows how to represent MRs in MiniZinc. Section V shows a case study. Next, Section VI carries out an evaluation of the case study. Section VII analyzes the threats to validity. Finally, the last section presents the conclusions and future lines of research.

II. Background

This section introduces the concepts necessary to understand the rest of the work. It briefly describes the MT technique and its main elements, as well as the most commonly used constraint satisfaction programs nowadays.

A. Metamorphic Testing

This technique has as fundamental elements the MRs, which are expected or existing properties on a series of different inputs and their corresponding results for multiple evaluations of an objective function [3].

We take as an example a function f, which given $m+1$ 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:

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

where $(n_0, n_1, n_2, ...n_m)$, is the original input and L_2 will be the following test case (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, a defect has been detected in f. 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.

Paper partially funded by the European Commission (FEDER) and the Spanish Government under the National Program for Research, Development and Innovation, Societal Challenges Oriented, Project DArDOS (TIN2015-65845-C3-3-R).

B. Constraint Programming Systems

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. Users have little to no control over this.

This paradigm has its beginnings in the 80s, although there is some prior work. For example, SketchPad (Sutherland, 1963) is an interactive constraint-driven drawing tool, and ALICE (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, CHIP V4 or ILOG 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 [10].

Constraint Programming Systems (CPS) are being used more often and in a wider range of domains ([11], [12]). CPSes are based on different programming languages and paradigms, and many different tools exist. Hakan provided a recent comparison of the available CPS in [13]. 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, Choco, Comet, Gecode, Tailor/Essence, Zinc, and MiniZinc were studied. Table I shows a fragment of that comparison.

Finally, the family of tools based on the Minizinc language has been chosen for the following reasons:

- Many solvers can parse MiniZinc.
- MiniZinc is easy to model with, providing high level elements and logical operators.
- There is plenty of documentation [14].
- It has a support community and recent updates.
- It is open source.

C. 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 in [15] 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 their approach using the existing INKA test case generator [16].

On the other hand, a recently published paper [17] proposes 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 [18]. They show that MT is effective in finding artificial bugs introduced through random code mutation.

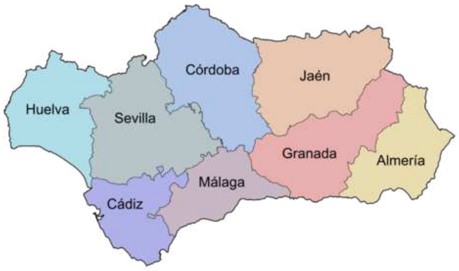

Fig. 1. Map of the provinces of Andalusia

```
% Colouring Andalusia using this many colours
int: nc = 3;

var 1..nc: al;  var 1..nc: ca;  var 1..nc: co;
var 1..nc: gr;  var 1..nc: hu;  var 1..nc: ja;
var 1..nc: ma;  var 1..nc: se;

constraint al != gr; constraint gr != ja;
constraint gr != co; constraint gr != ma;
constraint ja != co; constraint co != ma;
constraint co != se; constraint ma != ca;
constraint ma != se; constraint se != ca;
constraint se != hu;

solve  satisfy ;

output ["al=\(al)\t␣ca=\(ca)\n",
        "co=\(co)\t␣gr=\(gr)\t␣<hu=\(hu)\n",
        "ja=\(ja)\t␣ma=\(ma)\n",
         "se␣=", show(se),  "\n"];
```

Listing 1. Sample MiniZinc model

III. The MiniZinc language

MiniZinc is an open source constraint modeling language [19]. It can be used to model constraint satisfaction and optimization problems regardless of the resolver to be used. The default distribution of MiniZinc includes a graphical interface for ease of use, as well as examples [20]. The data file is separate from the model file. Both are translated to obtain a FlatZinc model, which depends on the specific solver used.

A MiniZinc model consists of a set of variable and parameter declarations, followed by a set of constraints. Model files use the extension *mzn*, while data files use *dzn*.

To illustrate the different parts of the model, a simple example is described in which the aim is to colour the map of the 8 provinces of Andalusia of Figure 1 with three different colours, so that the adjacent provinces have a different colour.

As we see in the example of Listing 1, the variable declaration region is clearly set apart from the constraint definition region. Optionally, a part where custom output formats can be defined is allowed.

TABLE I
COMPARISON OF THE SYSTEMS, SUBJECTIVE FEATURE MATRIX (RANGE: 1..5 WHERE 5 IS VERY GOOD, 1 IS NOT GOOD (OR N/A))

Feature	MiniZinc	Comet	Choco	JaCoP	Gecode	Tailor/Essence	Zinc
Ease of modelling	5	5	3	3	4	4	5
Documentation,site	3	4	4	4	4	3	4
Num. examples	4	4	3	4	4	3	3
Active community	2	5	4	2	4	1	2
Open source	5	1	5	5	5	3	4

```
Compiling andalucia.mzn
Running andalucia.mzn
al=2      ca=3
co=3      gr=1     <hu=2
ja=2      ma=2
se =1
----------
Finished in 17msec
```

Listing 2. Output produced by MiniZinc for sample model

The first line is a comment, beginning with the symbol `%`. Then, `int: nc = 3;` is a parameter definition. In the example, it represents the number of possible colors, as described in the example are 3. Parameters are allowed in which the type must be indicated. Supported types are: integer numbers (**int**), floating point numbers (**float**), Boolean (**bool**) and strings (**string**). It also supports arrays and sets.

The following lines that begin with `var` are variable declarations. In addition to parameters, decision variables are supported, which do not need to be initialized, but are given values when the constraint set is executed. However, it is necessary to define the domain of these variables, that is, the set of values they can take. As we see in the example, each variable refers to one of the provinces of Andalusia and all can be set to any number between 1 and the value of the parameter previously defined (`nc`), which represented the number of possible colors.

Decision variables can be integers, Boolean values, floating point numbers, or sets. It is also possible to define arrays whose elements are decision variables.

Constraints specify the logical expressions that the decision variables must satisfy in a valid solution to the model. They are statements like `constraint al != gr;` that begin with `constraint`. Relational operators are commonly used in constraints: equal (=, ==), not equal (!=), greater than (>), less than (<), greater than/equal (>=), and less than/equal (<=).

The `solve` statement indicates the type of problem (satisfaction or minimization/maximization). For example, `solve satisfy` is written for constraint satisfaction problems. Optionally, an `output` statement can define what to print once a solution has been found.

Once the model is written, it must be compiled, either in the command line or in the graphical interface. The example above produces the output in Listing 2. This output includes the compiled and executed files, the values for the decision variables, and the execution time.

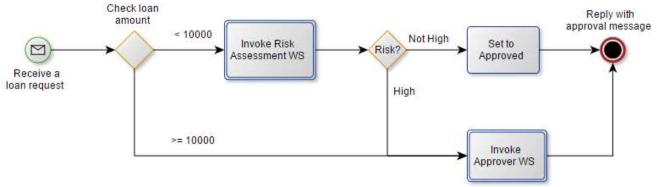

Fig. 2. Loan Approval BPMN diagram

IV. MAPPING MR TO MINIZINC

In this section we try to describe the representation of the MRs as a set of constraints in MiniZinc language. For this purpose, MRs as defined in Section II involve both the initial and the follow-up test cases. At a high level, the MiniZinc program will be made up of two sections:

- Decision variable and parameter declarations are used to define each value of interest for the initial and follow-up test cases. The declarations are typed according to the requirements of the test cases.
  Whether to use decision variables or parameters depends on the intended use of the program. The initial test and follow-up test can both be inputs (parameters) or outputs (decision variables) independently. The most typical scenario, which would be generating a follow-up test case from an initial one, would use parameters for the initial test and decision variables for the follow-up test. The case study will highlight other setups for different purposes.
- A set of constraints over the previously defined decision variables and parameters. Some of the constraints may be unique to the MR itself, while others may be related to variable/parameter domains or the context of the problem. These domain- or problem-related constraints are shared among all MRs in the same program. MiniZinc allows for modularity, so it is possible to extract all the shared declarations and problem-based constraints into a separate file, and include those from the MR-specific file. This prevents code duplication.

V. CASE STUDY

This section will present a case study on the use of MiniZinc to model the MR of a program. The program is a Web Service (WS) composition that is common in the WS literature: the

LoanApproval composition. This implements a new WS on top of other two: an *assessor* service and an *approver* service. The Business Process Model and Notation (BPMN) diagram for the composition is shown in Figure 2. The version of this program is described in [21]. The customer requests a loan of a certain amount: depending on the amount and the estimated risk, the composition decides whether it is granted or not.

As can be seen in the diagram on Figure 2, there is a threshold to decide whether an amount is "high". In this composition it is 10,000. There are two cases:

- For a low amount (< 10,000), the assessor service will be asked to estimate the risks associated to granting the loan. There are two possible answers:
  - "Low" risk loans are approved straight away (final reply is "true").
  - "High" risk loans require further checks and are passed to the approver service, who has the final say. The composition will approve the loan (answering "true") if the approver answers with "true", and will deny the loan (answering "false") otherwise.
- For a high amount ($\geq$ 10,000), the approver is called directly and has the final say. The assessor does not influence the execution in this case.

Based on the requirements of the composition, an initial set of test cases has been established, and several MRs have been proposed that the composition should meet. The following sections will be devoted to describing, formalizing and using these test cases and MRs through MiniZinc.

A. Metamorphic relations

We recall that each MR relates an initial test case (here it is represented by 1) to a subsequent test case (here it is represented by 2). The elements of each test case are:

- `req_amountN` is the amount requested by the customer, N can be 1 or 2 (depending on the initial case or the next case, generated by the MR).
- `ap_replyN` is the response from the approver service ("true" if approved, "false" if not).
- `as_replyN` is the assessor's response ("true" is high, "false" if low).
- `accepted` is the final answer given to the customer.

Given these elements, we designed the following MRs by considering the conditions that activated each of the available execution paths, and how the perturbation of one of the values would propagate to the others in each scenario (potentially even changing the activated branch):

- **MR1.1** (increasing high amount):
$$req\_amount2 = req\_amount1 * 10 \wedge req\_amount1 >= 10000 \wedge\ ap\_reply2 = ap\_reply1 \wedge\ as\_reply2 = as\_reply1 \implies accepted2 = accepted1$$
- **MR1.2** (increasing low amount):
$$req\_amount2 = req\_amount1 * 10 \wedge\ req\_amount1 < 10000 \wedge\ req\_amount2 < 10000 \wedge\ ap\_reply2 = ap\_reply1 \wedge\ as\_reply2 = as\_reply1 \implies accepted2 = accepted1$$
- **MR1.3** (going from low to high amount):
$$req\_amount2 = req\_amount1 * 10 \wedge\ req\_amount1 < 10000 \wedge\ req\_amount2 >= 10000 \wedge\ ap\_reply2 = ap\_reply1 \wedge\ as\_reply1 = true \implies accepted2 = accepted1$$
- **MR2.1** (reducing high amount):
$$req\_amount2 = req\_amount1/2 \wedge\ req\_amount1 >= 10000 \wedge\ req\_amount2 >= 10000 \wedge\ ap\_reply2 = ap\_reply1 \wedge\ as\_reply2 = as\_reply1 \implies accepted2 = accepted1$$
- **MR2.2** (reducing low amount):
$$req\_amount2 = req\_amount1/2 \wedge\ req\_amount1 < 10000 \wedge\ req\_amount2 < 10000 \wedge\ ap\_reply2 = ap\_reply1 \wedge\ as\_reply2 = as\_reply1 \implies accepted2 = accepted1$$
- **MR3** (negating the approver on a high amount):
$$req\_amount2 = req\_amount1 \wedge\ req\_amount1 >= 10000 \wedge\ ap\_reply2 = not(ap\_reply1) \wedge\ as\_reply2 = as\_reply1 \implies accepted2 = not(accepted1)$$
- **MR4** (negating the assessor on a high amount):
$$req\_amount2 = req\_amount1 \wedge\ req\_amount1 >= 10000 \wedge\ ap\_reply2 = ap\_reply1 \wedge\ as\_reply2 = not(as\_reply1) \implies accepted2 = accepted1$$

It can be observed that in MR1.1, MR1.2, MR2.1 and MR2.2 the following cases are similar to the initial cases, except that an arithmetic operation has been performed on the amount (multiply by a number in MR1.X, divide by an integer in MR2.X).

On the other hand, in MR3 a logic operation (negation) has been applied to the approver's response, leaving the rest of the data of the following case equal to those of the initial test case. Note that the restriction that the amount be greater than 10,000 is specified.

In MR4, the advisor's response has been modified by applying a logical operation (negation), leaving the rest of the data of the following case equal to those of the initial test case. As in the previous case, the amount must be greater than 10,000 to make sense according to the composition diagram.

The following section codifies some of the MRs described in MiniZinc language.

B. Mapping to MiniZinc

In this section, we are going to introduce the previous MRs in MiniZinc, and prove their validity.

We note that the types of test case data to be run are common to all MRs. It is also observed that there are restrictions that have to do with composition, therefore they are also common to all MRs. This common part is extracted to a separate file, which will be included from each MR file.

Listing 3 represents the common part of all MRs. The four elements in Section V-A are listed for the old and new test cases: the initial test case is a collection of parameters that will be set through the `.dzn` file, and the follow-up test case is a collection of decision variables that will contain

```
% PARAMETERS (OLD TEST CASE)
int: req_amount1;
bool: ap_reply1;
bool: as_reply1;
bool: accepted1;

% VARIABLES (NEW TEST CASE)
var int: req_amount2;
var bool: ap_reply2;
var bool: as_reply2;
var bool: accepted2;

%% COMPOSITION−SPECIFIC CONSTRAINTS
constraint req_amount1 >= 0;
constraint req_amount2 >= 0;
```

Listing 3. Common MiniZinc model for LoanApproval

```
include "loanAp_common.mzn";

%% MR−SPECIFIC CONSTRAINTS
constraint req amount1 < 10000 ;
constraint req amount2 < 10000 ;
constraint req_amount2 = req_amount1 * 10;
constraint ap_reply2 = ap_reply1;
constraint as_reply2 = as_reply1;
constraint accepted2 = accepted1;
```

Listing 4. MiniZinc model of MR1.2 (increasing low amount) of LoanApproval

MiniZinc's solution to the defined constrains. There are some basic constraints about the valid domain for the variables: loan amounts must be non-negative.

This common part is shared across all MRs. Listings 4 to 6 show the MiniZinc mappings for MR1.2, MR3 and MR4. The mapping from the original logic was relatively straightforward. However, mapping constraints on more complex data structures may prove to be trickier: this will have to be evaluated in more complex case studies.

```
include "loanAp_common.mzn";

%% MR−SPECIFIC CONSTRAINTS
constraint req_amount1 >= 10000 ;
constraint req_amount2 = req_amount1;
constraint ap_reply2 = not ap_reply1;
constraint as_reply2 = as_reply1;
constraint accepted2 = not accepted1;
```

Listing 5. MiniZinc model of MR3 (negating the approver on a high amount) of LoanApproval

```
include "loanAp_common.mzn";

%% MR−SPECIFIC CONSTRAINTS
constraint req_amount1 >= 10000 ;
constraint req_amount2 = req_amount1;
constraint ap_reply2 = ap_reply1;
constraint as_reply2 = not as_reply1;
constraint accepted2 = accepted1;
```

Listing 6. MiniZinc model of MR4 (negating the assessor on a high amount) of LoanApproval

```
% VARIABLES (OLD TEST CASE)
var int: req_amount1;
var bool: ap_reply1;
var bool: as_reply1;
var bool: accepted1;

% VARIABLES (NEW TEST CASE)
var int: req_amount2;
var bool: ap_reply2;
var bool: as_reply2;
var bool: accepted2;

%% COMPOSITION−SPECIFIC CONSTRAINTS
constraint req_amount1 >= 0;
constraint req_amount2 >= 0;
```

Listing 7. Common MiniZinc model of LoanApproval with variables

C. Checking satisfiability

A minimum requirement that every MR must meet is that it is not a logical contradiction. For example, an MR could not require that a variable be smaller and larger than 10 at a time. This check can be automated with MiniZinc by leaving all identifiers in the program as decision variables. The MiniZinc solver will find at least one case meeting all restrictions.

This is done by modifying the file of Listing 3 with the common information, leaving it as shown in Listing 7. Some examples of outputs are shown for some of the MRs cited above. For MR1.2, we see the output generated by the system in the Listing 8: the MR modified the requested amount by multiplying it by a number (in this case 10).

It is interesting to see that the outputs of Listing 8 are not valid test cases: when the risk is low (`as_reply = false`) the credit should be approved (`accepted = true`). MRs depend on the validity of their initial test cases, and the original values (ending in "1") did not represent an original case. In this step we have only checked that MR1.2 is satisfiable: in the following section we will check whether given an original valid test case, it can produce a new valid test case as well.

All MRs were tested in this way, and they produced at least one solution. Therefore, they were found to be satisfiable. In the following section, the usual use of MRs to generate new valid test cases from other valid test cases is shown.

```
Compiling LoanApMR1_2.mzn
Running LoanApMR1_2.mzn
req_amount1 = 1
ap_reply1 = false
as_reply1 = false
accepted1 = false
req_amount2 = 10
ap_reply2 = false
as_reply2 = false
accepted2 = false
----------
Finished in 46msec
```

Listing 8. MiniZinc output of MR1.2 of LoanApproval with variables

TABLE II
GENERATED TESTS, WITH SOURCE MRS (HIGH AMOUNT)

MR	Initial test case	Following test case
`MR1.1`	(15000, true, true, true)	(150000, true,true, true)
`MR2.1`	(15000, true, true, true)	(7500, true,true, true)
`MR3`	(15000, true, true, true)	(15000, false,true, false)
`MR4`	(15000, true, true, true)	(15000, true, false, true)

D. Generating new cases

Once the satisfiability of the proposed MRs has been checked, this section tries to use these MRs to generate new test cases (the follow-up test cases). These new test cases extend the original test suite, and can detect errors which were not picked up before.

This is the traditional use of MRs in the MT technique. In MiniZinc, the common model file will use parameters for the values of the original test, and variables for the following test. The solver will compute the values the variables should take, based on the values given for the parameters.

Three initial test cases were selected by looking at the various paths available in Figure 2:

- For high amounts, `(amount = 15000, ap_reply = true, as_reply = false, accepted = true)` was used. While the common model file requires a value for `as_reply`, this value did not impact executions: as seen in Figure 2, the assessor is not used for large amounts. Table II lists the generated test cases and the MRs which produced them.
- For low amounts, there are two options: low risk or high risk. For low risk, `(amount = 150, ap_reply = false, as_reply = false, accepted = true)` could be used with MR1.2 and MR2.2. For high risk, `(amount = 15000, ap_reply = false, as_reply = true, accepted = false)` was used with MR1.3. Table III lists the generated test cases.

In short, the initial suite with three test cases was extended with seven new test cases. The next section will check if these new test cases resulted in a more effective test suite.

TABLE III
GENERATED TESTS, WITH SOURCE MRS (LOW AMOUNT)

MR	Initial test case	Following test case
`MR1.2`	(150, false, false, true)	(1500, false,false, true)
`MR2.2`	(150, false, false, true)	(75, false, false, true)
`MR1.3`	(1500, false, true, false)	(15000, false, false, false)

VI. EVALUATION

After generating an extended test suite, the next step was to see if these new test cases are useful or not: specifically, whether they can detect defects where the original tests could not. To do this, the MuBPEL mutation testing tool [22] was used. MuBPEL implements the mutation operators defined by Estero-Botaro et al. [23], [24], which produce *mutants* of the original program with small changes — these may introduce defects that a good test suite should pick up.

The following steps were performed with MuBPEL:

1) The composition was analyzed, finding the locations where each mutation operator can be applied.
2) All possible mutants were generated.
3) The original composition was executed against each initial test case, collecting its outputs.
4) The mutants were executed against the same initial test cases, and their outputs are compared against those of the original composition. If they produced different outputs, they were said to be *killed* and were discarded for the rest of the procedure.
5) The original composition was executed against each following test cases, collecting its outputs.
6) The remaining mutants were executed against the following test cases, and their outputs were compared against those of the original composition.

Useful MRs will produce tests that kill mutants (i.e. defects) which the initial tests could not. On the other hand, if the MRs fail to kill any more mutants we can either expand the set of MRs, or try again with an expanded set of initial tests.

In this case study, the initial test suite consists of the three test cases shown in the second column of Tables II and III. Table IV collects the results of applying mutation testing on Loan with the initial and following test suites.

MuBPEL produced 98 mutants from the Loan composition, using 33 mutation operators. The initial test suite killed 50 mutants, and 2 of the mutants failed to deploy: 46 mutants survived the initial test suite. The following test suite killed 34 of the remaining mutants, leaving 12 mutants still alive.

These results show that the initial test suite has been improved through the above MRs: the expanded test suite (the combination of initial and following tests) was able to detect defects that the initial test suite could not.

Closer inspection of the 12 surviving mutants shows that 5 of them simply require very specific values of the `amount` variable. One option would be to complement metamorphic testing with a technique that took into account constant values in the program. Another option would be adding a more

TABLE IV
MuBPEL validation results

Suite	Total	Killed	Alive	# tests
Initial	96	50	46	3
Follow-up	96	84	12	3 + 7

specific MR which changed the amount, and an initial test case with an amount closer to the condition threshold (10,000).

The remaining 7 surviving mutants can be considered *equivalent*: manual inspection of the differences did not show any meaningful changes in behaviour.

VII. Threats to Validity

The results in Section VI show that our approach can automate the generation of new test cases by reusing an existing constraint solver, and that the new 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).

A. Internal threats

The initial test cases were created by hand, making sure that there was one 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.

Seven 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.

In this work, the follow-up test cases have been evaluated by checking if they could kill the surviving mutants. Ideally, the metamorphic relations themselves should have been used as test oracles to check the quality of the follow-up test cases. In future work, we will change the experimental procedure to use MRs as test oracles, as the concept of MRs would normally suggest.

It would have been useful to check if simply generating more test cases randomly would have produced similar results. However, the focus of this study was on providing a guided approach for test generation: it is unlikely that random generation would scale up to more complex programs under test. We plan to add this evaluation to future studies with larger programs.

B. External threats

The chosen case study (the LoanApproval composition) is a small one. 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 to be used.

While the selected case study used scalar value types (integers and booleans), there are extensions to MiniZinc that allow for more complex data types, such as vectors or character strings. 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.

In this first study, the designed MRs are all based on input-only and output-only relations, as described by Chen [3]. However, looking at Figure 2, it would have been possible to define an MR where the amount was under 10000 and the risk was high: in that case, the reply from the composition (an output) would have been that same as the reply from the approver (an input). In future work, we will study further this other type of MRs in more complex scenarios, using constraint solvers.

VIII. Conclusion and future work

Metamorphic testing (MT) has seen considerable advances in the recent years, but still has some pending challenges. One of them is the definition, design and implementation of metamorphic relations (MRs) in a formal and uniform manner. The present work has proposed a way to represent MRs through MiniZinc, a popular constraint programming language for which many solvers are available.

A case study has been presented to show the feasibility of the approach, with MRs and initial test cases. With MiniZinc, it was possible to both check the internal logical consistency of the MRs and have them generated the following test cases, with only minor changes to the constraint models. The effectiveness of the following test cases was checked through mutation testing: the following test cases were able to detect 34 defects that were not picked up by the initial test cases.

As future work to further improve defect detection, several possibilities are raised: including new test cases, including new MRs that generate new test cases, iterate the process, or using the test cases generated as initial test cases in a subsequent iteration of the process.

The proposed approach has given promising initial results in this small case study. However, further evaluations need to be performed with larger and more complex programs, and the efficiency and effectiveness of the MRs must be studied. Beyond expanding the set of case studies, future work will include a systematic approach to elicit and specify MRs through constraint programming, providing additional support to users of metamorphic testing.

References

[1] T. Chen, S. Cheung, and S. Yiu, "Metamorphic testing: A new approach for generating next test cases," *HKUSTCS98-01*, 1998.

[2] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[3] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 4:1–4:27, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3143561

[4] U. Kanewala, L. L. Pullum, S. Segura, D. Towey, and Z. Q. Zhou, Eds., *Proceedings of the IEEE/ACM 1st International Workshop on Metamorphic Testing (MET '16), in conjunction with the 38th International Conference on Software Engineering (ICSE '16)*. ACM, 2016, ISBN 978-1-4503-4163-9.

[5] L. L. Pullum, D. Towey, U. Kanewala, C. A. Sun, and M. E. Delamaro, Eds., *2nd ACM/IEEE International Workshop on Metamorphic Testing (MET 2017)*. ACM, May 2017, ISBN 978-1-5386-0424-3.

[6] X. Xie, L. L. Pullum, and P. L. Poon, Eds., *3rd ACM/IEEE International Workshop on Metamorphic Testing (MET 2018)*. ACM, 2018, ISBN 978-1-4503-5729-6.

[7] T. Y. Chen, P.-L. Poon, and X. Xie, "METRIC: METamorphic relation identification based on the category-choice framework," *Journal of Systems and Software*, vol. 116, pp. 177 – 190, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215001624

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

[9] S. Segura, A. Duran, J. Troya, and A. R. Cortes, "A template-based approach to describing metamorphic relations," in *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, May 2017, pp. 3–9. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MET.2017.3

[10] J. Jaffar and M. J. Maher, "Constraint logic programming: A survey," *The Journal of Logic Programming*, vol. 19, pp. 503–581, 1994.

[11] K. Apt, *Principles of Constraint Programming*. Cambridge University Press, 2003.

[12] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of Constraint Programming*. Elsevier, 2006.

[13] H. Kjellerstrand, "Comparison of CP systems," Available from http://www.it.uu.se/research/group/astra/SweConsNet12/hakan.pdf, 2012, date of last access: 10 February 2019.

[14] P. J. Stuckey, K. Marriott, and G. Tack, "MiniZinc handbook release 2.2.3," https://www.minizinc.org/doc-2.2.3/en/MiniZincHandbook.pdf, 2018, date of last access: 15 March 2019.

[15] A. Gotlieb and B. Botella, "Automated metamorphic testing," in *27th International Computer Software and Applications Conference (COMPSAC 2003): Design and Assessment of Trustworthy Software-Based Systems, 3-6 November 2003, Dallas, TX, USA, Proceedings*, 2003, pp. 34–40. [Online]. Available: https://doi.org/10.1109/CMPSAC.2003.1245319

[16] A. Ingenierie, "Thales airborne systems," *INKA–V1 User's Manual*, 2002.

[17] Ö. Akgün, I. P. Gent, C. Jefferson, I. Miguel, and P. Nightingale, "Metamorphic testing of constraint solvers," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2018, pp. 727–736.

[18] I. P. Gent, C. Jefferson, and I. Miguel, "Minion: A fast scalable constraint solver," in *ECAI*, vol. 141, 2006, pp. 98–102.

[19] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a standard CP modelling language," in *Principles and Practice of Constraint Programming – CP 2007*, C. Bessière, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 529–543.

[20] Data61 research network, CSIRO, "MiniZinc," https://www.minizinc.org/index.html, 2018, date of last access: 18 February 2019.

[21] UCASE Software Engineering research group, "WS-BPEL composition repository," https://neptuno.uca.es/redmine/projects/wsbpel-comp-repo/wiki/LoanApprovalDoc, 2018, date of last access: 26 December 2018.

[22] UCASE research group, "MuBPEL wiki page," https://neptuno.uca.es/~mubpel, Aug. 2015, date of last access: 18 March 2019.

[23] J. Boubeta-Puig, A. García-Domínguez, and I. Medina-Bulo, "Analogies and Differences between Mutation Operators for WS-BPEL 2.0 and Other Languages," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Berlin, Germany: IEEE, Mar. 2011, pp. 398–407.

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