Automatic Discovery and Cleansing of Numerical Metamorphic Relations

Anonymous Authors

***Abstract*—Metamorphic relations (MRs) describe the invariant relationships between program inputs and outputs. By checking for violations of MRs, faults in programs can be detected. Identifying MRs manually is a tedious and error-prone task. In this paper, we propose AutoMR, a novel method for systematically inferring and cleansing MRs. AutoMR can discover various types of equality and inequality MRs through a search method (particle swarm optimization). It also employs matrix singular-value decomposition and constraint solving techniques to remove the redundant MRs in the search results. Our experiments on 37 numerical programs from two popular open source packages show that AutoMR can effectively infer a set of accurate and succinct MRs and outperform the state-of-the-art method. Furthermore, we show that the discovered MRs have high fault detection ability in mutation testing and differential testing.**

***Index Terms*—Metamorphic relations, program invariants, search-based method, metamorphic testing.**

I. Introduction

Metamorphic relations (MRs) [1]–[6] describe the invariant relationships between program inputs and outputs. For example, when testing a program which calculates the sin value for an input, it is not obvious for testers to determine the expected value of an arbitrary input. However, testers can test the programs by leveraging the mathematical property of the sin function such as $sin(x) + sin(-x) = 0$ and $sin(x) - sin(x + 2\pi) = 0$. These properties are MRs that should hold for all inputs of the program. In other words, testers can indirectly test the implementation by investigating whether the inputs and outputs satisfy the MRs. In this way, the program can be tested without knowing the correct outputs for arbitrary inputs. MRs have been used in metamorphic testing [7]–[11] to test programs that do not have easy-to-obtain test oracles [12], such as scientific programs, search engines, bioinformatics programs, etc.

In current practice MRs are mainly identified in an ad-hoc manner [6], which requires testers to have solid knowledge about the program and the problem domain. Recently, several studies have been conducted to automatically identify MRs [7], [13]–[16]. However, these works are still in early stage [6]. For example, Zhang et al. [16] proposed an approach to automatically infer two types of polynomial MRs, the kind of MRs which can be expressed as polynomials. They developed a software solution called MRI (Metamorphic Relation Inferrer) and evaluated the inferred MRs by detecting bugs in mutation testing. Although effective, their approach was limited to only two types of equality MRs and a significant number of the inferred MRs were redundant [16]. The identification and selection of high-quality MRs are still recognized as a big challenge [6].

In this paper, we present AutoMR, a novel approach to automatic identification of various types of high-quality MRs. In our approach, we first propose a general parameterization of arbitrary polynomial MRs, including both equalities and inequalities. We then adopt the PSO (particle swarm optimization) technique to search for suitable parameters for the MRs. Finally, with the help of matrix SVD (singular-value decomposition) and constraint solving techniques, we cleanse the MRs by removing the redundancy. In this way, we obtain a set of accurate and succinct MRs. The discovered and cleansed MRs are important program invariants that are useful for program understanding and maintenance. They are also useful for software testing, especially regression testing, mutation testing, and differential testing [17].

We have applied our approach to 37 numerical programs from NumPy [18] and Apache Commons Math [19], and inferred eight types of equality and inequality MRs. To evaluate the ability of MRs in software testing, we utilized the MRs to test 625 mutated programs (i.e., 625 seeded faults). The results show that in total 374 mutants were killed (i.e., faults were successfully revealed) by the MRs inferred by AutoMR, nearly 4 times as many as the those detected by the state-of-the-art approach MRI [16]. We also use the MRs inferred from the NumPy programs to discover faults seeded in Apache programs that have the same specifications. The results confirm the effectiveness of MRs in differential testing [17].

The main contributions of this paper include:

- A general method to infer both equality and inequality polynomial MRs of arbitrary degree. To our best knowledge, this is the first attempt to automatically infer inequality MRs.
- A novel approach integrating matrix SVD and constraint solving techniques to remove redundant MRs.
- An empirical study to demonstrate the MRs inferred by AutoMR were able to detect bugs and outperform the state-of-art approach.

The organization of this paper is as follows. In Section II, we present the background and motivation of MR identification. In Section III, we present our approach and an example to illustrate the ability of AutoMR. We design our experiment in Section IV and present the results in Section V. In Section VI we discuss the results and analyze the threats to validity. Section VII discusses the related work and Section VIII concludes this paper.

II. Background and Motivation

Identifying MRs manually is a tedious and error-prone task. Although several studies have been conducted to automatically identify MRs [7], [13], [14], [16], the research is still at its early stage. The MRI approach proposed by Zhang et al. [16] is the current state-of-art. MRI can discover linear and quadratic equality MRs. It first parameterizes two predefined types of MRs, then searches for the suitable parameters by multiple executions of the program, and finally filter away false MRs by conducting more executions. However, the following issues have not been addressed by previous works:

① *Inequality MRs.* MRs are often mistaken for just equality relations [6]. However, inequality MRs are also able to hold important information on programs' behaviors. For example, $exp(x+1) - exp(x) > 0$ is an important MR of the program exp, as this MR describes its monotonicity.

② *MRs involving more inputs.* Usually MR is considered to involve only two inputs and their corresponding outputs. An example of such case is the MR $sin(x)+sin(-x) = 0$, which describes the relation of two inputs x and $-x$ and their outputs. However, there also exist some MRs that involve three or more inputs and their outputs, which can describe more complex logic of the program. For example, a more complicated MR, $sin(2x) - 2sin(x)sin(\pi/2 - x) = 0$, involves three inputs, i.e., x, $2x$ and $\pi/2 - x$.

③ *Arbitrary-degree MRs.* MRI allows the input relation to be linear and the output relation to be linear or quadratic, thus not able to identify relations of higher degree. For example, for the program of sin, MR $sin(x) + sin(-x) = 0$ could be found but $sin^3(x) - 3sin(x) - 4sin(3x) = 0$ which involves third degree could not.

④ *Redundant MRs.* The MRs obtained by search-based methods contain an amount of redundancy, resulting in extra efforts on verifying and applying MRs. For example, after manually checking the inferred MRs of a program, Zhang et al. found that only 2 out of 219 were unique and all the other 217 MRs were redundant [16].

In this paper, we propose a novel automatic MR discovery approach that can overcome the above limitations. Our approach can infer arbitrary polynomial MRs and support inequality MRs, as well as remove the redundant ones.

III. Proposed Approach

In this section, we describe the proposed approach to automatic discovery of MRs for numerical programs. Figure 1 illustrates the overall workflow of AutoMR. First, we design a general method for parameterizing arbitrary polynomial equality or inequality MRs (Section III-A) and adopt PSO to search for the parameters based on a set of randomly generated test inputs (Section III-B). Then another set of test inputs is randomly generated and applied to the search results to filter away the false MRs (Section III-C). Next, we employ SVD and constraint solving techniques to detect the redundant MRs (Section III-D), thus obtaining succinct MRs that can be used for program understanding and testing.

A. Parameterizing arbitrary equality and inequality polynomial MRs

1) Definitions: Without loss of generality, for a program under test P, we assume that:

- The program takes the input $\mathbf{i}$ and produces the corresponding output $\mathbf{o}$:
$$P(\mathbf{i}) = \mathbf{o}$$
- Each input $\mathbf{i}$ has n elements:
$$\mathbf{i} = \begin{pmatrix}\mathbf{i}_1 & \mathbf{i}_2 & \dots & \mathbf{i}_n\end{pmatrix}^T$$
- Each output $\mathbf{o}$ has m elements:
$$\mathbf{o} = \begin{pmatrix}\mathbf{o}_1 & \mathbf{o}_2 & \dots & \mathbf{o}_m\end{pmatrix}^T$$

We define a polynomial MR in terms of input relation and output relation:

- The MR involves h inputs and their corresponding outputs. The inputs are denoted as $\mathbf{i}^{(1)}, \mathbf{i}^{(2)} \dots \mathbf{i}^{(h)}$, the outputs are denoted as $\mathbf{o}^{(1)}, \mathbf{o}^{(2)} \dots \mathbf{o}^{(h)}$.
- The inputs comply with a k-degree polynomial relation, denoted as $\mathbb{R}_{input}$.
- Their corresponding outputs comply with a l-degree polynomial relation, denoted as $\mathbb{R}_{output}$.

With the above definitions, an MR of P can be described as: *If h inputs, $\mathbf{i}^{(1)}, \mathbf{i}^{(2)} \dots \mathbf{i}^{(h)}$, satisfy the input relation $\mathbb{R}_{input}$, their corresponding outputs, $\mathbf{o}^{(1)}, \mathbf{o}^{(2)} \dots \mathbf{o}^{(h)}$, will comply with the output relation $\mathbb{R}_{output}$.*

We describe the details about parameterizing the MRs in the following subsections. We will also give an example to explain the concepts.

2) Input relation: Firstly, we consider the equality input relations. $\mathbf{i}^{(1)}$ is selected as the base input and we construct a vector $\mathbf{u}$, which comprises the products of the combinations for the elements of the base input, from 0-degree (constant term) to k-degree (highest-degree term). As $\mathbf{u}$ contains all the possible terms of the polynomials, in the following process we can just focus on finding the appropriate coefficients for these terms. The number of elements of $\mathbf{u}$ is denoted as t, where

$$t = 1 + n + \frac{(n+2-1)!}{2! \times (n-1)!} + \dots + \frac{(n+k-1)!}{k! \times (n-1)!} \quad (1)$$

To satisfy $\mathbb{R}_{input}$, any of the inputs other than $\mathbf{i}^{(1)}$ can be expressed as $\mathbf{i}^j = \mathbf{A}^{(j-1)}\mathbf{u}, \quad j = 2, 3 \dots h$, where $\mathbf{A}^{(j-1)}$ is a $(n \times t)$ matrix which we aim to find. Then the $\mathbb{R}_{input}$ can be described as:

$$\mathbb{R}_{input} : \mathbf{i}^j = \mathbf{A}^{(j-1)}\mathbf{u}, \quad j = 2, 3 \dots h \quad (2)$$

Next, we extend the parameterization to inequality input relations. Inequalities can be either greater than or less than relations, as shown in the following equations:

$$\mathbb{R}_{input} : \mathbf{i}^j > \mathbf{A}^{(j-1)}\mathbf{u}, \quad j = 2, 3 \dots h \quad (3)$$

$$\mathbb{R}_{input} : \mathbf{i}^j < \mathbf{A}^{(j-1)}\mathbf{u}, \quad j = 2, 3 \dots h \quad (4)$$

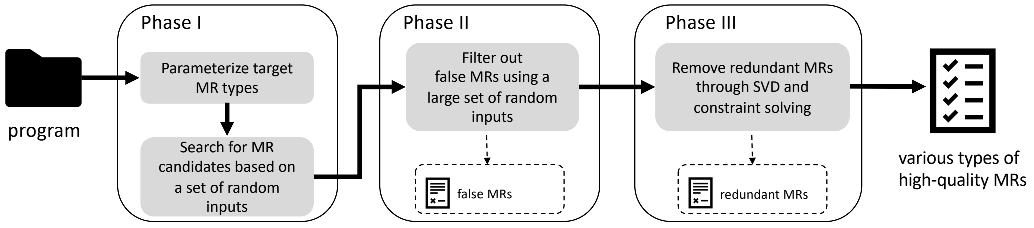
program
Phase I
Parameterize target MR types
Search for MR candidates based on a set of random inputs
Phase II
Filter out false MRs using a large set of random inputs
false MRs
Phase III
Remove redundant MRs through SVD and constraint solving
redundant MRs
various types of high-quality MRs

Fig. 1: The overall workflow of AutoMR

Because we need to feed the program concrete inputs to return outputs, we introduce an additive random matrix to transfer the inequality relations to equality relations, as shown in Equation 5. In the equation, ϵ is an additive matrix containing randomly generated values. The elements of the input are kept in their domain after adding the random values, which is ensured by dynamically generating ϵ according to the gaps between the elements and their boundaries.

$$\mathbb{R}_{input} : \mathbf{i}^{j} = \mathbf{A}^{(j-1)}\mathbf{u} + \epsilon, \quad j = 2, 3 \dots h \tag{5}$$

3) Output relation: We construct a matrix $\mathbf{o}^{(1:h)}$, which consists of the elements of all the outputs from $\mathbf{o}^{(1)}$ to $\mathbf{o}^{(h)}$: $\mathbf{o}^{(1:h)} = (\mathbf{o}^{(1)}\ \mathbf{o}^{(2)} \dots \mathbf{o}^{(h)})$. With the assumption that outputs comply with a l-degree polynomial relation, a matrix $\mathbf{v}$ is introduced here, which consists of all the polynomial combinations of the elements of $\mathbf{o}^{(1:h)}$, from 0-degree (constant term) to l-degree (highest-degree term). We denote the number of elements of $\mathbf{v}$ is r (e.g. Equation 15), where

$$r = 1 + hm + \frac{(hm+2-1)!}{2! \times (hm-1)!} + \cdots + \frac{(hm+l-1)!}{l! \times (hm-1)!} \tag{6}$$

The output relation, $\mathbb{R}_{output}(\mathbf{o}^{(1)}, \mathbf{o}^{(2)} \dots \mathbf{o}^{(h)})$, will hold if we can find a matrix $\mathbf{B}$ so as to satisfy:

$$\mathbf{Bv} = 0 \tag{7}$$

It is worth noting that we denote an inferred MR as equality if its left part is equal or close to 0 (e.g., the absolute value is smaller than 0.05).

For inequality relations, $\mathbb{R}_{output}$ could be:

$$\mathbb{R}_{output} : \mathbf{Bv} > 0 \tag{8}$$

$$\mathbb{R}_{output} : \mathbf{Bv} < 0 \tag{9}$$

4) Combine the parameterized input and output relations: After parameterizing the input and output relations, the process of inferring MRs is transferred to find pairs of matrices $\mathbf{A}$ and $\mathbf{B}$ that satisfy both the input and output relations. In this way, each pair of $\mathbf{A}$ and $\mathbf{B}$ describes an MR of program P.

5) An Example: Consider a program under test P, which calculates the $sine$ value of a given number. Equations 10 and 11 are two MRs of P that we aim to discover.

$$\mathbf{MR1}: \quad sin(2x) - 2sin(x)sin(\pi/2 - x) = 0 \tag{10}$$

$$\mathbf{MR2}: \quad sin(x) - sin(2\pi + x) = 0 \tag{11}$$

The above two MRs are equalities and the process of identifying these MRs are as follows. As there is only one element in each input, we denote the base input as $x1$. Based on the parameterization of input relations, we assume the input relations can be denoted as:

$$\mathbf{u} = \begin{pmatrix} 1 & x1 \end{pmatrix}^{T} \tag{12}$$

$$x2 = \mathbf{A}^{(1)}\mathbf{u} \tag{13}$$

$$x3 = \mathbf{A}^{(2)}\mathbf{u} \tag{14}$$

where $\mathbf{A}^{(1)}$ and $\mathbf{A}^{(2)}$ are (2×1) matrices that we aim to find.

For the output relation, matrix $\mathbf{v}$ can be denoted as :

$$\mathbf{v} = \begin{pmatrix} 1 \\ sin(x1) \\ sin(x2) \\ sin(x3) \\ sin^2(x1) \\ sin^2(x2) \\ sin^3(x3) \\ sin(x1)sin(x2) \\ sin(x1)sin(x3) \\ sin(x2)sin(x3) \end{pmatrix} \tag{15}$$

The output relation can be denoted as:

$$\mathbf{B}v = 0 \tag{16}$$

where $\mathbf{B}$ is a (10×1) matrix that we aim to find.

Based on the assumptions, the pairs of matrices $\mathbf{A}$ and $\mathbf{B}$ actually describe MRs for program P. With the help of search method (to be described in Section III-B), we can find pairs of appropriate matrices, such as:

$$\mathbf{MR1}: \mathbf{A}^{(1)} = \begin{pmatrix} 0 & 2 \end{pmatrix} \tag{17}$$

$$\mathbf{A}^{(2)} = \begin{pmatrix} \pi/2 & -1 \end{pmatrix} \tag{18}$$

$$\mathbf{B} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -2 & 0 \end{pmatrix} \tag{19}$$

$$\mathbf{MR2}: \mathbf{A}^{(1)} = \begin{pmatrix} 2\pi & 1 \end{pmatrix} \tag{20}$$

$$\mathbf{B} = \begin{pmatrix} 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{21}$$

The above pairs of matrices $\mathbf{A}$ and $\mathbf{B}$ actually describe the MRs shown in Equation 10 and 11.

*B. Phase I: Search for matrices **A** and **B***

To find the matrices $\mathbf{A}$ and $\mathbf{B}$ that satisfy the relations, we use a metaheuristic algorithm called Particle Swarm Optimization (PSO). PSO is a widely-used swarm intelligence optimization algorithm [20], [21]. In PSO, each candidate solution is called a particle, and multiple particles coexist and optimize cooperatively to achieve the optimal solutions. PSO is a proper way of discovering MRs as there can be more than one MRs for a program, correspondingly there exist more than one optimal solutions.

In the search process, each pair of matrices $(\mathbf{A}, \mathbf{B})$ is treated as one particle. The performance of a particle is evaluated by the cost function, as shown in Equation 22. The smaller of a position's cost, the nearer it is to the optimal. In the equation, s is the number of test inputs that are randomly generated. For equality MRs, the cost of a particle, $\sum_{z=1}^{s} |\mathbf{B}\mathbf{v}^{(z)}|$, can be regarded as its accumulated distances to zero for all test inputs. For inequality MRs, q is the number of the passed test inputs, so the cost of a particle is the proportion of failed test inputs against it.

$$cost(\mathbf{A}, \mathbf{B}) = \begin{cases} \text{for equality MRs:} & \sum_{z=1}^{s} |\mathbf{B}\mathbf{v}^{(z)}| \\ \text{for inequality MRs:} & 1 - \frac{q}{s} \end{cases} \tag{22}$$

At the beginning of the search, we randomly generate an initial population of particles as candidates. After that, during the searching iterations all particles keep updating their positions according to their own costs and the global best cost. After the iterations, the position of the particle which has the best performance during all the moments will be returned as the result.

C. Phase II: Filter out the false MRs

In order to accelerate the search process, we use a small number of test inputs for evaluating the cost function in Phase I. To increase the reliability of identified MRs, in Phase II we randomly generate a much larger set of random test inputs and use it to filter away the false MRs produced in Phase I. The MRs which fail in this validation are discarded. In our experiments, we initially generate 100 random test input in Phase I and then generate another 100 random test inputs for filtering in Phase II. The filtering process is repeated 100 times and the remaining MRs are passed to Phase III.

D. Phase III: Remove redundant MRs

There exists redundancy in the discovered MRs. For example, PSO may find the following three MRs for the sin program: (i) $sin(x)+sin(-x) = 0$, (ii) $sin(x)-sin(x+2\pi) = 0$ and (iii) $sin(-x)+sin(x+2\pi) = 0$. However, the third MR is redundant because it can be inferred by the combination of the first and second. This phenomenon has been observed in MRI's results, in which only 2 out of 219 MRs are representative after manually checking [16].

The redundancy also exists in inequality MRs. For the program log, after Phase II we might get these two MRs: (i) $if\ x^{(2)} > x^{(1)},\ log(x^{(2)}) - log(x^{(1)}) > 0$ and (ii) $if\ x^{(2)} > x^{(1)} + 2,\ log(x^{(2)}) - log(x^{(1)}) + 1 > 0$. The second MR is redundant as it could be inferred from the first one.

After investigating the results from Phase II, we find that for equality MRs most redundancy is related to their multi-collinearity, whereas for inequality MRs it is related to the strictness of the relations, as shown in above examples. To remove the redundant MRs, we propose an algorithm that takes advantage of matrix singular-value decomposition (SVD) and constraint solving techniques (Figure 2).

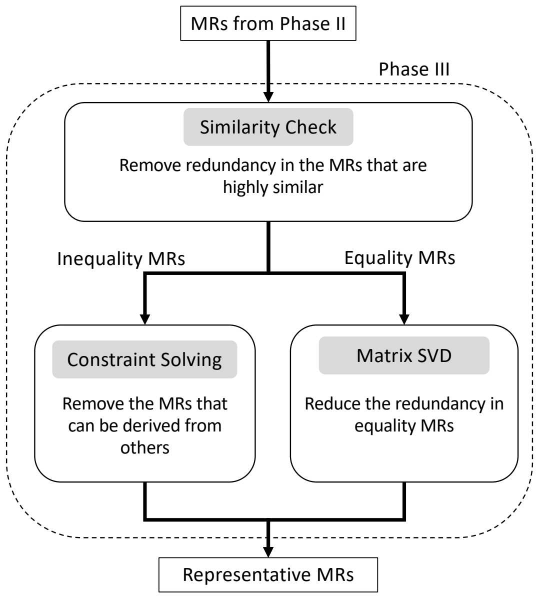
MRs from Phase II
Phase III
Similarity Check
Remove redundancy in the MRs that are highly similar
Inequality MRs
Equality MRs
Constraint Solving
Remove the MRs that can be derived from others
Matrix SVD
Reduce the redundancy in equality MRs
Representative MRs

Fig. 2: The workflow of Phase III

Similarity check. The first step is to remove the redundant MRs that are highly similar. For example, due to the inaccuracy of the search process, we observed two MRs: $(i) sin(x) - sin(x+2\pi) = 0$ and $(ii) 2.05sin(x) - 1.98sin(x+2.03\pi) = 0$. It can be seen that the second MR is redundant of the first, though they are not exactly the same. In order to handle this type of redundancy, we introduce a metric to evaluate two parameterized MRs' similarity. It is a normalized Euclidean distance, shown as Equation 23. In the equation, M_{ij} denotes the element of i^{th} row and j^{th} column of the matrix, r_{ij} denotes the searching range of M_{ij}, n is the number of elements of M. The calculated distance is in the range of $[0, 1]$ and a smaller value indicates that the two matrices are more similar. The distance will be 0 when two matrices are exactly the same. The threshold to determine the high-similarity can be user-defined and in our experiments 0.05 is adopted empirically. For any two MRs, if both their input relations and output relation matrices are highly similar, we deem one of the MRs redundant.

$$distance(M^{(1)}, M^{(2)}) = \frac{\sqrt{\sum_{i,j} \left(\frac{M_{ij}^{(1)} - M_{ij}^{(2)}}{r_{ij}} \right)^2}}{n} \tag{23}$$

Matrix SVD. For equality MRs, we employ the matrix SVD technique to reduce the redundancy among MRs.

Firstly, we construct a matrix $\mathbf{C}$ in which each row represents an MR. In this way, the problem of obtaining the representative MRs is transferred to finding a maximal linearly independent subset of the matrix $\mathbf{C}$. After that, each row of the maximal linearly independent subset will be an unique MR. Usually, the way of getting the maximal linearly independent subset is to transfer the matrix to its row echelon form. However, due to the inaccuracy of the search results, the matrix cannot be transferred to a useful row echelon form. To solve this problem, we make use of the SVD method in our approach. Firstly, we treat each MR (i.e., each row) as a feature and the corresponding values as the data of the feature. In this way, the problem of finding distinct MRs is further transferred to reducing the number of features. The SVD method can factorize the matrix $\mathbf{C}$ to the product of three matrices $\mathbf{USV^T}$ and $\mathbf{S}$ is a rectangular diagonal matrix with non-negative numbers on the diagonal, known as the singular values. Next, we can discard the redundant features which correspond to the very small singular values. In AutoMR, this algorithm is implemented based on a machine learning package, scikit-learn [22].

For example, we might have these MRs after Phase II for the program P. $x1$, $x2$, $x3$ are three inputs that comply with some input relations:

(i) $P(x1) + 0.99P(x2) = 0$
(ii) $P(x1) + 1.01P(x3) = 0$
(iii) $P(x2) - 1.03P(x3) = 0$
(iv) $1.98P(x1) + 0.99P(x2) + 1.01P(x3) = 0$

Note that equality MRs here only mean that their left parts are close to 0 (smaller than 0.05). It can be seen that these four MRs contain redundancy, because (iii) and (iv) can be inferred from (i) and (ii). As described above, the matrix $\mathbf{C}$ for the three outputs $P(x1)$, $P(x2)$, $P(x3)$ will be:

$$\begin{pmatrix} 1 & 0.99 & 0 \\ 1 & 0 & 1.01 \\ 0 & 1 & -1.03 \\ 1.98 & 0.99 & 1.01 \end{pmatrix}$$

After applying the SVD transformation, we can get the following matrix:

$$\begin{pmatrix} 2.433 & 1.198 & 1.252 \\ -0.030 & -1.235 & 1.238 \\ -0.004 & 0.004 & 0.004 \end{pmatrix}$$

It can be computed that the singular values of the rows are $(0.24, 0.76, 0)$, which means that the first two rows contain enough information of the matrix. Therefore, we can remove the redundancy and reconstruct the semantically equivalent MRs using the first two rows:

(i) $2.433P(x1) + 1.198P(x2) + 1.252P(x3) = 0$
(ii) $-0.030P(x1) - 1.235P(x2) + 1.238P(x3) = 0$

Constraint Solving. For inequality output relations, we can also construct a matrix $\mathbf{C}$, in which each row represents the output relation of an MR. However, due to the reason that elementary row operations will not keep the original inequality of the relation, we are not able to use SVD to remove the redundancy. For example, the SVD method can't detect that $log(x+1) - log(x) + 1 > 0$ is redundant to $log(x+1) - log(x) > 0$. To handle such cases, we make use of the constraint solving technique in our approach. For any two inequality MRs (i) and (ii), if (i) has a more lenient input relation and meanwhile a more stringent output relation than (ii), we deem that the MR (ii) is redundant of (i). For example, in the following two MRs, (i) $if\ x^{(2)} > x^{(1)},\ P(x^{(2)}) - P(x^{(1)}) > 0$ and (ii) $if\ x^{(2)} > x^{(1)} + 1,\ P(x^{(2)}) - P(x^{(1)}) + 1 > 0$, MR (i)'s input relation is more lenient and output relation is more stringent than (ii), so MR (ii) is redundant. In AutoMR, we integrate Z3 [23], a constraint solver from Microsoft, to check the strictness of the relations. Z3 is a SMT (Satisfiability modulo theories) based solver which can check the satisfiability of user-provided formulas. For any two relations $R1$ and $R2$, we represent them symbolically in the constraint solver and check their satisfiability, as shown in the following equation:

$$\left.\begin{array}{l} (R1 \text{ AND } R2) \text{ is satisfiable} \\ (\neg R1 \text{ AND } R2) \text{ is unsatisfiable} \\ (R1 \text{ AND } \neg R2) \text{ is satisfiable} \end{array}\right\} \Rightarrow R1 \text{ is lenient}$$

E. An Example

Suppose we have implemented a program P, which evaluates the sin value (output) for a given input. In order to obtain MRs, we may use an existing MR inference technique such as MRI [16]. However, due to the limitations of MRI, only two certain types of MRs can be obtained and the results contain quite a number of redundant MRs, as shown in Table I. The MRs in bold font are the representative ones, such as (a) $P(x) - P(x+2\pi) = 0$, (c) $P(x) + P(x+\pi) = 0$ and (g) $(P(x))^2 + (P(-x+\pi/2))^2 - 1 = 0$. Because of the inaccuracy of the search results, redundant MRs such as (b) $2.05P(x) - 1.98P(x+2\pi) = 0$ and (h) $2.03(P(x))^2 + 1.98(P(-x+\pi/2))^2 - 1.95 = 0$ are also found. MR (f) $P(x+\pi) + P(x+2\pi) = 0$ is also a kind of redundancy, because it can be derived by the combination of (a) and (b). A false MR, (e) $P(x) - P(x-2) - 1.5 = 0$, might also be discovered, though it will be filtered out in the following filtering phase.

AutoMR can infer more kinds of MRs and alleviate the pain to obtain the representative MRs. As shown in Table I, the inferred MRs (a) to (h) are the same as MRI. However, AutoMR can also infer MRs (i) to (p), which cannot be discovered by MRI. Among them, MR (i) $P(x) + P(x+2) - 2.05 < 0$ and (m) $P(x) + P(x-3) + 2.05 > 0$ are inequality relations. MR (o) $P(2x) - 2P(x)P(\pi/2 - x) = 0$ involves three inputs, and MR (p) $P(3x) - 3P(x) + 4(P(x))^3 = 0$ is a cubic polynomial relation. Actually, the user could set arbitrary degrees to the input and output relations for the MRs to be inferred. Moreover, in Phase III, the redundant MRs such as

TABLE I: Example MRs of *sin* inferred by MRI and AutoMR

Approach	Phase I: Inferred MRs	Phase II: MRs after filtering	Phase III: MRs after removing redundancy
MRI	**(a) $P(x) - P(x + 2\pi) = 0$** (b) $2.05P(x) - 1.98P(x + 2\pi) = 0$ **(c) $P(x) + P(x + \pi) = 0$** (d) $1.45P(x) + 1.47P(x + \pi) = 0$ (e) $P(x) + P(x-2) - 1.5 = 0$ (f) $P(x + \pi) + P(x + 2\pi) = 0$ **(g) $(P(x))^2 + (P(-x + \pi/2))^2 - 1 = 0$** (h) $2.03(P(x))^2 + 1.98(P(-x + \pi/2))^2 - 1.95 = 0$ ...	**(a) $P(x) - P(x + 2\pi) = 0$** (b) $2.05P(x) - 1.98P(x + 2\pi) = 0$ **(c) $P(x) + P(x + \pi) = 0$** (d) $1.45P(x) + 1.47P(x + \pi) = 0$ (f) $P(x + \pi) + P(x + 2\pi) = 0$ **(g) $(P(x))^2 + (P(-x + \pi/2))^2 - 1 = 0$** (h) $2.03(P(x))^2 + 1.98(P(-x + \pi/2))^2 - 1.95 = 0$ ...	Not Supported
AutoMR	(a) ~ (h): Same as MRI's **(i) $P(x) + P(x + 2) - 2.05 < 0$** (j) $P(x) + P(x + 2) - 3 < 0$ (k) $P(x) + P(x + 5) - 1.5 > 0$ (l) $P(x) + P(x + 5) - 1 > 0$ **(m) $P(x) + P(x - 3) + 2.05 > 0$** (n) $P(x) + P(x - 3) + 2.3 > 0$ **(o) $P(2x) - 2P(x)P(\pi/2 - x) = 0$** **(p) $P(3x) - 3P(x) + 4(P(x))^3 = 0$** ...	(a)(b)(c)(d)(f)(g)(h): Same as MRI's **(i) $P(x) + P(x + 2) - 2.05 < 0$** (j) $P(x) + P(x + 2) - 3 < 0$ **(m) $P(x) + P(x - 3) + 2.05 > 0$** (n) $P(x) + P(x - 3) + 2.3 > 0$ **(o) $P(2x) - 2P(x)P(\pi/2 - x) = 0$** **(p) $P(3x) - 3P(x) + 4(P(x))^3 = 0$** ...	**(a) $P(x) - P(x + 2\pi) = 0$** **(c) $P(x) + P(x + \pi) = 0$** **(g) $(P(x))^2 + (P(-x + \pi/2))^2 - 1 = 0$** **(i) $P(x) + P(x + 2) - 2.05 < 0$** **(m) $P(x) + P(x - 3) + 2.05 > 0$** **(o) $P(2x) - 2P(x)P(\pi/2 - x) = 0$** **(p) $P(3x) - 3P(x) + 4(P(x))^3 = 0$** ...

* MRs in **bold** indicate that they are not redundant.

$(b)(d)(h)(f)$ will be filtered out so that a much smaller set of high-quality MRs can be obtained.

IV. Experimental Design

A. Research Questions

RQ1: Is AutoMR able to identify equality and inequality MRs?

AutoMR is designed to be a general technique that can handle an arbitrary degree of equality and inequality polynomial MRs. The first RQ is to evaluate its effectiveness in identifying MRs of some real-world numerical programs. Besides, we also evaluate AutoMR's ability to remove redundancy for both equality and inequality MRs.

RQ2: Can the discovered MRs be used to detect program faults?

As MRs contain behavioral properties of the program, the second RQ is to investigate whether such information can be used to reveal program faults. In this RQ, we conduct mutation testing and differential testing experiments to evaluate the fault-detection ability of the inferred MRs.

B. Subject Programs

To evaluate the proposed approach, we applied AutoMR on 8 programs from Apache Commons Math 2.2 and 29 programs from NumPy 1.14. Apache Commons Math [19] and NumPy [18] are commonly used math libraries for Java and Python. The list of the programs is shown in Table II. The collection comprises several types of programs, including unary-variate input and output programs such as *sin* and *log1p*, multiple-variates input and/or output programs such as *hypot* and *sort*. The 8 Apache programs were also used in related work [16] so that we could conduct a comparative study. To systematically evaluate the MRs' ability to detect faults, we applied the MRs on 8 Apache programs' 625 mutants, which were also used for evaluating MRI [16]. Each mutant, which is the result of applying a mutation operator to the source code, is viewed as a faulty program in our study.

TABLE II: Subject Programs

Apache Commons Math 2.2	NumPy 1.14
abs, asinh, atan, cos, log1p, log10, sin, tan	array_equal, dot, abs, arccos, arccosh, arcsin, arcsinh, arctan, arctanh, ceil, arctan2, cos, cosh, hypot, exp, floor, amax, log, log1p, amin, log10, round, sin, sinh, sqrt, tan, tanh, power, sort

C. Experiment Settings

1) RQ1: We used our approach to infer MRs for 29 programs from NumPy and 8 programs from Apache Commons Math, then compared the results with [16]. For each of the subject programs, PSO was run 500 times with maximum iterations of 350 for each type of MRs. For each run of the PSO, the population of the test inputs was 100. The number of the initial particles was set to 20, and the input domain of the programs was set between 0 and 20. The searching boundaries of the coefficients were set as -10 to 10 for constant terms, -2 to 2 for non-constant terms. After obtaining the MRs from the PSO runs, we applied the filtering process to filter out the false MRs. That was done by testing the MRs with 100 randomly generated test inputs and repeated 100 times.

2) RQ2: In this RQ, mutation testing and differential testing strategies were employed to evaluate the fault-detection ability of the inferred MRs. Mutation testing is to seed bugs to the original program to generate a large corpus of faulty programs (mutants) that can be used for systematically evaluating a test set's fault-detection ability [24]. Differential testing is to compare the executions of similar programs (or different implementations of the same specification) and observe the difference [17]. In our experiment, MRs inferred from the 8 Apache programs were used to test their 625 mutants (mutation testing). Besides, MRs inferred from the NumPy programs which have the same specifications as the 8 Apache programs were also used to test the mutants (differential testing). If an MR killed a mutant while passed the original program, that would be recorded as a true detection. But if an MR killed both the mutant and the original program, that would be recorded as a false detection.

As the MRs are inferred by repetitively executing the program on the test inputs in Phase I, we also directly use these test inputs together with the corresponding program outputs to kill the mutants and compare the mutant killing rates with those achieved by the MRs.

V. Experimental Results

A. RQ1: Inference of various types of MRs

Table III presents the numbers of various types of MRs inferred by our approach. The results are categorized into 8 types according to the properties of their input and output relations. We can see from the table that for each type our approach managed to infer a number of MRs. For the studied programs, the average number of discovered MRs ranges from 4 to 409. By manually checking some of the results, we find that the results reveal some important properties of the programs. For example, AutoMR discovered that $sin(x + 2\pi) - sin(x) = 0$ and $sin(-x) + sin(x) = 0$, which shows that sin is a periodical and odd function. There also exist some complex MRs such as $1.35cos(x) + 1.49cos(1.37x + 6.97) - 4.45 < 0$. Though this MR is not very intuitive, it implies important information: the output of the program has a boundary.

Type I and II are the two types of MRs supported by MRI, the approach proposed in [16]. For the 8 programs from Apache Commons Math, we compared the numbers of Type I and II MRs inferred by AutoMR and MRI, shown in the last two columns of Table III. For the 8 programs, on average MRI obtained 107 Type I and 37 Type II MRs. Comparatively, after removing the redundancy, AutoMR only obtained 6 and 11 MRs. Though less MRs were obtained, we observed that Type I and II MRs inferred by our approach could detect more faults than the ones inferred by MRI (shown in Figure 3). Such results indicate that the MRs inferred by MRI contained a lot of duplicates which did not contribute to the detection of program faults.

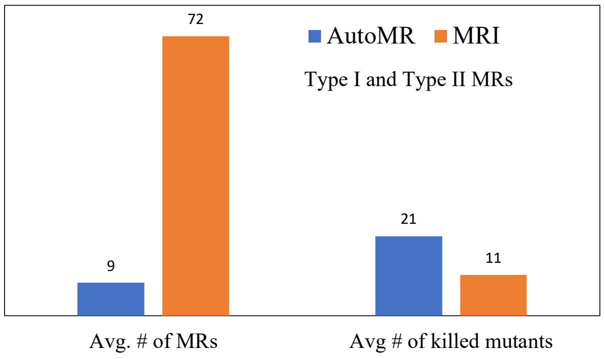

Fig. 3: Comparison of AutoMR and MRI on inferring Type I and II MRs

The higher mutant killing rate of MRs inferred by AutoMR can be explained by its novel cost function in Phase I. While for equality MRs MRI sets criteria and counts the number of passed test inputs, in our approach AutoMR evaluates a particle's cost by summing all test inputs' distance to zero. When trying MRI's cost function in our experiment, we found that during the PSO iterations there may exist more than one candidates that possessed the same cost. In such a situation MRI has to select one of them as the global best and thus may miss the true best direction of the search process. In our approach the total distance for different particles could rarely be the same. Therefore, at each moment there would be only one global best particle. Compared with the cost function of MRI, the optimal solution is more likely to be reached at the end of our search process.

In this RQ, we also investigate the ability of AutoMR on removing redundancy. Table IV presents the redundancy removal rates for equality and inequality MRs of the 8 Apache programs. On average, AutoMR removed 19.7% and 75.0% redundant MRs for inequality and equality MRs respectively. We also applied the redundancy removal process on the two types of MRs inferred by MRI in [16] and managed to remove a significant amount of redundancy. On the one hand, the redundancy removal rates indicate the existence of redundant MRs in the search results, which was observed but not addressed in [16]. On the other hand, the results show the effectiveness of our redundancy removal algorithm. Compared to redundancy removing by human [16], we take advantages of constraint solving technique and matrix properties to save manual efforts.

B. RQ2: Fault-detection capacity of the inferred MRs

Table V summarizes the numbers of the mutants (faulty programs) killed by the MRs inferred from the original (unmutated) Apache programs. It can be seen that the inferred MRs successfully detected a number of faults and did not report any false detections. The highest killing rate happened on tan, for which 88.9% of the faults were detected. The lowest detection rate was on $asinh$, the inferred MRs still detected 23.6% of the faults. The average mutant killing rate for the 8 programs was 50.6%, suggesting that the inferred MRs successfully can record behavioral properties of the original programs and can be used in software testing.

Furthermore, to evaluate the effectiveness of MRs in differential testing [17], we tested the mutated Apache programs using the MRs inferred from the NumPy programs that have the same specifications. On average, the fault detection rate was 51.3% and the number of false detection was 0, indicating that the MRs inferred from NumPy still hold for the corresponding implementations in Apache. We can also see from Table V that the fault detection rates of MRs inferred from NumPy were similar to the ones from Apache. This is because that the programs from these two packages possess the same specifications.

Compared with MRI, we observed that more types of MRs could enhance the fault detection capacity, as shown in Figure 4. It can be seen that the MRs inferred by AutoMR can detect many more faults than those inferred by MRI.

TABLE III: MR inference results

Type	Input relation	Output relation	# of inputs	Degree of input relation	Degree of output relation	Example*	Avg. # of MRs		
							29 NumPy programs (supported by AutoMR)	8 Apache programs (supported by both AutoMR and MRI)	
							AutoMR	AutoMR	MRI
I	equality	equality	2	1	1	$x2 = x1 + \pi$; $\sin(x2) + \sin(x1) = 0$	18	6	107
II	equality	equality	2	1	2	$x2 = 2x1 - \pi$; $\cos(x2) + 2\cos^2(x1) - 1 = 0$	16	11	37
III	equality	equality	2	1	3	$x2 = 0.5x1 + 2.25\pi$; $2.4\sin(x1) - 0.98\sin(x2) + 1.63\sin^2(x1) - 1.01\sin(x1)\sin(x2) - 1.43\sin^2(x2) - 0.07\sin^3(x1) + 0.05\sin^2(x1)\sin(x2) - 3.27\sin(x1)\sin^2(x2) + 1.95\sin^3(x2) + 0.71 = 0$	11	7	Not Supported
IV	equality	equality	3	1	1	$x2 = x1 + 3\pi$ and $x3 = x1 - 3\pi$; $\cos(x1) - 0.5\cos(x2) - 0.5\cos(x3) = 0$	115	80	
V	equality	equality	3	1	2	$x2 = 2x1 + 0.5\pi$ and $x3 = -x1 - 3\pi$; $-0.3\sin(x1) - 0.5\sin(x2) + 0.3\sin(x3) - 0.6\sin^2(x1) + 1.2\sin(x1)\sin(x2) + 0.3\sin(x1)\sin(x3) - 1.2\sin(x2)\sin(x3) - 0.7\sin^2(x3) + 0.5 = 0$	57	61	
VI	inequality	equality	2	1	1	$x2 > 2x1 + 3.82$; $0.08\operatorname{atan}(x1) - 1.54\operatorname{atan}(x2) + 2.25 = 0$	11	4	
VII	equality	inequality	2	1	1	$x2 = 1.37x1 + 6.97$; $1.35\cos(x1) + 1.49\cos(x2) - 4.45 < 0$	409	349	
VIII	inequality	inequality	2	1	1	$x2 > 1.24x1 + 6.13$; $-1.36\operatorname{floor}(x1) + 1.79\operatorname{floor}(x2) - 0.11 > 0$	386	395	

* Value close to 3.14 is denoted as π. A relation is regarded as equality left part of the formula is always less than a small value, i.e. 0.05 in this table.

TABLE IV: Effectiveness of removing redundancy

Program	Equality MRs				Inequality MRs	
	Inferred by AutoMR (Type I-V)		Inferred by MRI (Type I, II)		Inferred by AutoMR (Type VI-VIII)	
	# of MRs before/after redundancy removal	Redundancy removal rate	# of MRs before/after redundancy removal	Redundancy removal Rate	# of MRs before/after redundancy removal	Redundancy removal rate
abs	157/50	68.2%	350/25	92.9%	349/313	10.2%
asinh	74/31	58.6%	146/2	98.6%	363/313	13.9%
atan	218/58	73.4%	23/2	91.3%	464/329	29.1%
cos	320/78	75.6%	76/6	92.1%	333/300	9.7%
log1p	141/20	85.7%	315/22	93.0%	369/289	21.8%
log10	153/27	82.4%	56/2	96.4%	363/277	23.7%
sin	306/79	74.1%	151/2	98.7%	333/307	7.9%
tan	21/4	82.5%	28/13	53.6%	39/23	41.4%

For example, among the 197 faulty versions of $asinh$, MRs inferred by AutoMR can successfully detect 23.6% of them, whereas only 0.3% were detected by the MRs inferred by MRI. This improvement could be attributed to the various types of MRs which rely on different branches of the program. AutoMR could infer not only MRs supported by MRI, but also other complex equality as well as inequality MRs. The results suggest that it is not the quantity but the diversity of MRs that contributes more to the fault detection ability. The importance of the MRs' diversity was also highlighted by Liu et al. [4]. They applied MRs to verify the correctness of a program and found that using more diverse MRs could enhance the cost-effectiveness of metamorphic testing.

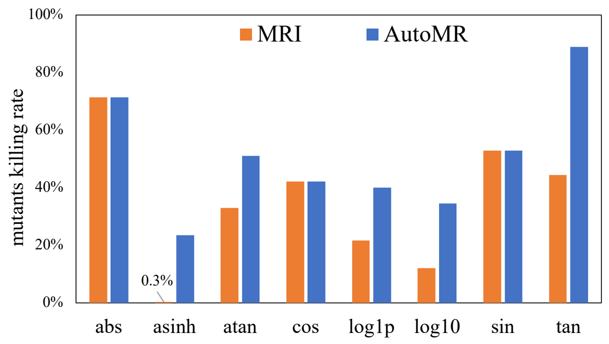

Fig. 4: Fault detection capacity of AutoMR and MRI

Figure 5 shows the numbers of faulty programs that were detected using MRs and concrete test cases whose inputs were used to infer the MRs. It can be seen that MRs inferred by AutoMR could detect nearly two times as many as bugs detected by the concrete test cases. The reason is that during the process of inferring MRs, the cost function guides the program under test to execute repetitively to reduce the cost. Consequently, the inferred MRs contain more behavioral information than the original test cases. As a comparison, the MRs inferred by MRI killed less mutants than the concrete test cases, which

TABLE V: Detection of seeded faults using inferred MRs

Program	# of seeded faults	Detected by MRs from Apache										Detected by MRs from NumPy
		# TD by Type I	# TD by Type II	# TD by Type III	# TD by Type IV	# TD by Type V	# TD by Type VI	# TD by Type VII	# TD by Type VIII	# TD by All Types	Total TD rate	Total TD rate
abs	14	10	10	10	10	10	10	10	10	10	71.4%	71.4%
asinh	297	1	1	1	3	1	64	1	5	70	23.6%	25.3%
atan	188	70	66	70	76	66	84	54	56	96	51.1%	51.1%
cos	38	16	16	0	16	16	0	16	16	16	42.1%	42.1%
log1p	230	54	0	0	54	56	88	52	58	92	40.0%	39.1%
log10	116	8	8	8	8	8	36	8	12	40	34.5%	22.4%
sin	34	18	18	18	18	18	0	18	18	18	52.9%	58.8%
tan	36	16	16	0	16	16	0	16	30	32	88.9%	100.0%

* False Detection was not observed for all programs.
* TD denotes True Detection, i.e. the original program passed but the mutant was killed.

indicated that part of the information in the test cases was missing.

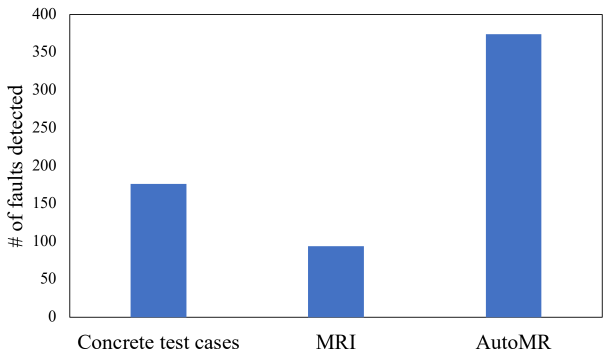

Fig. 5: Number of faults detected by three approaches (total number of faults is 625). (i) detected by concrete test cases whose inputs were used to infer MRs. (ii) MRI: detected by MRs inferred by MRI. (iii) AutoMR: detected by MRs inferred by AutoMR.

Table VI compares the performance of AutoMR and MRI on MR inference and bug detection. Despite the fact the MRs inferred by AutoMR could detect nearly four times as many bugs as MRI, AutoMR spent slightly more time than MRI. On average, MRI needed 3.9 s to infer an MR and detect a bug but AutoMR needed 4.7 s. This discrepancy might be explained by two reasons: firstly, AutoMR infers more complex relations such as the cubic degree MRs. Secondly, AutoMR has an additive cleansing phase to remove the redundant MRs. Above all, regarding the significantly increased fault-detection ability, we think the extra consumed time is worthwhile.

VI. Discussions

A. Discussion of Results

From the experimental results, it can be seen that the inferred MRs are able to represent program properties and reveal faulty behaviors. However, although PSO could find some solutions to our problem, it cannot ensure to find all solutions and sometimes returns duplicates. Therefore, in our experiments we repeated PSO 500 times to find as many distinct results as possible. Previously, Zhang et al. [16] tried to manually identify the representative MRs from hundreds of search results. Compared to their approach, we integrate constraint solving technique and matrix properties to automate the process. This will save the tester much time and computing resources when the MRs are used for testing purposes.

TABLE VI: Performance of MR inference and bug detection

Performance	AutoMR	MRI
# of faults detected by MRs	94	374
Avg. time consumed for obtaining one MR and detecting one fault (seconds)	4.7	3.9

Although effective, AutoMR cannot detect all faults, as shown in the experiment results (Table V). The reason why some faults failed to be detected could be that the collection of inferred MRs was incomplete. As Chen et al. pointed out [6], MRs are necessary properties of the target algorithm in relation to multiple inputs and their expected outputs, and there are usually a huge number of these properties. For example, new MRs could be obtained by increasing the number of involved inputs and the degree of relations. In our future work, we will explore more effective techniques to discover more MRs.

To our knowledge, the proposed tool has the following applications:

- Facilitate program understanding. As our approach does not require the source code of the program, AutoMR can be used to help understand the programs whose source code are not available. In our experiment, without knowing the source code AutoMR succeeded to discover MRs that revealed the periodicity and boundedness of some trigonometric programs.
- Facilitate regression testing. In software maintenance and evolution, the MRs inferred from a certain version can be

used to ensure that a new version still works. Some subtle faults might not be identified by the programmers but can be tracked by the inferred MRs.

- Facilitate differential testing. MRs could be inferred from one program and used to test its alternative implementations. This is a strategy of differential testing [17]. For example, in our experiments, we use the MRs inferred from NumPy to test the Apache programs that follow the same specifications. Our experiment results have shown that with the help of MRs, more faults can be revealed than just using concrete test cases.

B. Threats to Validity

We have identified the following threats to validities:

- Subjects: In this study, we use a collection of 37 numerical programs. Although these programs are from two math packages (NumPy, Apache Commons Math) implemented in three different languages (C, Python and Java), the number of subject programs is still relatively small. In our future work, we will experiment with more programs to further evaluate the effectiveness of our approach.
- Numerical programs only: in this paper we only studied numerical MRs. There are other metamorphic relations among categorical values, strings, and objects, which will be investigated in our future work. However, we would like to stress that numerical software provides the foundation for a wide variety of scientific and safety-critical systems and whose failures have catastrophic consequences [25].
- Accuracy: In our experiments on detecting faults using MRs returned by AutoMR, no false detection (false alarm) was detected. However, as shown in our experimental results, our approach still failed to detect some seeded faults (Table V). We have discussed the possible reasons in Section VI-A. We will improve the proposed approach in our future work.

VII. Related Work

Many studies explore the invariants in programs and find the invariants useful in software maintenance [26], fault detection [27], [28], and fault localization [29]. A metamorphic relations is also a kind of invariant between multiple program inputs and outputs.

Metamorphic relations are at the core of metamorphic testing. Constructing metamorphic relations manually tends to be a tedious process and requires thorough knowledge about the program under test. In recent years, many studies have been conducted to automatically or semi-automatically construct metamorphic relations [13], [14], [16], [30]–[35]. For example, Liu et al. [32] proposed to construct metamorphic relations by combining existing metamorphic relations, whose approach is named CMR (composition of metamorphic relations). The core of CMR is to identify composable metamorphic relations, which depends on manual identification. Carzaniga et al. [33] proposed to generate test oracles by analyzing the redundancy in a program under test. Such redundancy can help construct metamorphic relations by replacing some operations in a test with redundant operations. Here they need to manually identify redundancy in a program. Kanewala and Bieman [14], [34] utilized machine learning techniques to predict whether metamorphic relations exist for a program in a set of predefined relations. Here they mainly used the control flow graph information of a method. Su et al. [35] proposed to dynamically infer likely metamorphic relations by searching a collection of predefined metamorphic relations. Zhang et al. [16] proposed a search-based metamorphic relation construction approach (MRI), which automatically infers two types of polynomial relations based on multiple executions of a program.

In this paper, we propose AutoMR, an automatic metamorphic relation construction approach. The most related work is MRI [16], since AutoMR also utilizes a search-based technique to infer polynomial metamorphic relations. Search-based software testing uses metaheuristic algorithms (e.g. genetic algorithm, hill climbing algorithm, PSO) to address software testing problems such as test case generation, selection, and prioritization [36]–[39], whereas our approach aims to discover MRs using metaheuristic algorithm. Compared with MRI, AutoMR addresses the limitations in the number of inputs, the degree of relations and equality relations. That is, AutoMR is a general approach to inferring polynomial relations involving equality and inequality relations, multiple-variates inputs and outputs. In particular, AutoMR utilizes matrix singular-value decomposition and constraint solving techniques to reduce redundant relations, while MRI does not deal with redundancy automatically.

VIII. Conclusion

In this paper, we propose AutoMR, a technique for automatically inferring and cleansing Metamorphic Relations (MRs). AutoMR can infer both equality and inequality MRs, and MRs of linear, quadratic, and even higher degrees. We have applied our approach to 37 numerical programs and evaluated the fault-detection capacity of the inferred MRs. The result shows that AutoMR can effectively infer various types of MRs and outperforms the state-of-art approach. Moreover, the MRs inferred by AutoMR were successfully used to help detect faults in the mutation testing and differential testing experiments. Our experimental tool and data are available at **https://github.com/bolz213/AutoMR**.

In the future, we will extend the proposed approach to support more types of MRs such as logical and categorical MRs. We will also investigate MR-based automatic bug detection and program repair.

References

[1] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, Tech. Rep. HKUST-CS98-01, 1998.

[2] T. Y. Chen, F. C. Kuo, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing and beyond," in *Eleventh Annual International Workshop on Software Technology and Engineering Practice*, pp. 94–100.

[3] Z. Zhou, D. Huang, T. Tse, Z. Yang, H. Huang, and T. Chen, "Metamorphic testing and its applications," in *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004), Software Engineers Association, Tokyo, Japan*. Software Engineers Association, 2004.

[4] H. Liu, F. C. Kuo, D. Towey, and T. Y. Chen, "How Effectively Does Metamorphic Testing Alleviate the Oracle Problem?" *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 4–22, Jan. 2014.

[5] T. Y. Chen, "Metamorphic Testing: A Simple Method for Alleviating the Test Oracle Problem," in *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*, pp. 53–54.

[6] 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.

[7] H. Zhu, "JFuzz: A Tool for Automated Java Unit Testing Based on Data Mutation and Metamorphic Testing Methods," in *2015 Second International Conference on Trustworthy Systems and Their Applications*, Jul. 2015, pp. 8–15.

[8] M. Jiang, T. Y. Chen, F.-C. Kuo, D. Towey, and Z. Ding, "A metamorphic testing approach for supporting program repair without the need for a test oracle," vol. 126, pp. 127 – 140.

[9] X.-W. Lv, S. Huang, Z.-W. Hui, and H.-J. Ji, "Test cases generation for multiple paths based on PSO algorithm with metamorphic relations."

[10] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. P. J. C. Bose, N. Dubash, and S. Podder, "Identifying Implementation Bugs in Machine Learning Based Image Classifiers Using Metamorphic Testing," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. ACM, pp. 118–128.

[11] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, "Metamorphic Testing of RESTful Web APIs," pp. 1–1.

[12] E. J. Weyuker, "On Testing Non-Testable Programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[13] 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.

[14] U. Kanewala and J. M. Bieman, "Using machine learning techniques to detect metamorphic relations for programs without test oracles," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, Nov. 2013, pp. 1–10.

[15] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, B. Xie, and H. Mei, "Supporting Oracle Construction via Static Analysis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 178–189.

[16] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei, "Search-based Inference of Polynomial Metamorphic Relations," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 701–712.

[17] W. M. McKeeman, "Differential testing for software," vol. 10, no. 1, pp. 100–107.

[18] "NumPy," *http://www.numpy.org*.

[19] "Commons Math: The Apache Commons Mathematics Library," *http://commons.apache.org/proper/commons-math*.

[20] M. Clerc, *Particle Swarm Optimization*. John Wiley & Sons, 2010, vol. 93.

[21] A. Windisch, S. Wappler, and J. Wegener, "Applying Particle Swarm Optimization to Software Testing," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '07. New York, NY, USA: ACM, 2007, pp. 1121–1128.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[23] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[24] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[25] A. D. Franco, H. Guo, and C. Rubio-González, "A comprehensive study of real-world numerical bug characteristics," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Oct. 2017, pp. 509–519, iSSN:.

[26] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," vol. 27, no. 2, pp. 99–123.

[27] R. Wang, Z. Ding, N. Gui, and Y. Liu, "Detecting Bugs of Concurrent Programs With Program Invariants," vol. 66, no. 2, pp. 425–439.

[28] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas, "AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 37. IEEE Computer Society, pp. 269–280.

[29] R. Abreu, A. González, P. Zoeteweij, and A. J. C. van Gemund, "Automatic Software Fault Localization Using Generic Program Invariants," in *Proceedings of the 2008 ACM Symposium on Applied Computing*, ser. SAC '08. ACM, pp. 712–717.

[30] G.-W. Dong, B.-W. Xu, L. Chen, C.-H. Nie, and L.-L. Wang, "Case studies on testing with compositional metamorphic relations," *Journal of Southeast University (English Edition)*, vol. 24, no. 4, pp. 437–443, 2008.

[31] U. Kanewala, J. M. Bieman, and A. Ben-Hur, "Predicting metamorphic relations for testing scientific software: A machine learning approach using graph kernels," *Software testing, verification and reliability*, vol. 26, no. 3, pp. 245–269, 2016.

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

[33] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè, "Cross-checking oracles from intrinsic software redundancy," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 931–942.

[34] U. Kanewala, "Techniques for automatic detection of metamorphic relations," in *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference On*. IEEE, 2014, pp. 237–238.

[35] F.-H. Su, J. Bell, C. Murphy, and G. Kaiser, "Dynamic inference of likely metamorphic properties to support differential testing," in *Proceedings of the 10th International Workshop on Automation of Software Test*. IEEE Press, 2015, pp. 55–59.

[36] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-oriented Software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. Szeged, Hungary: ACM, 2011, pp. 416–419.

[37] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on software engineering*, vol. 33, no. 4, 2007.

[38] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, "Empirical evaluation of pareto efficient multi-objective regression test case prioritisation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 234–245.

[39] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2013.