Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Evaluation of Automatically Generated Metamorphic Relations

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von:	Daniel Klodt
geboren am:	23.11.1999
geboren in:	Berlin
Gutachter/innen:	Prof. Dr. Lars Grunske
	Prof. Dr. Timo Kehrer
eingereicht am:	....
verteidigt am:	....

Contents

Abstract

In today's day and age, software keeps getting more powerful and therefore keeps getting more complex. To keep up with that, software testing has become an integral part in the software development process.
But how are we supposed to test a program, where we don't know the exact results beforehand?
This problem is also known as the oracle problem. Scientific software is one of the victims of the oracle problem. One way to bypass this has been metamorphic testing[CCY20], as the approach of it is, that you rather look at the relations between outputs based on the relation of the respective inputs, than at the input-output relations.

Müller et al. [Mül+22] have been developing a tool, that is automatically looking for these metamorphic relation(MRs) for the exiting-nomad parser. So far, the tool is only able to determine candidates for MRs, but is yet to evaluate the correctness and the effectiveness of them.

This thesis will carry on from this point. We will look at the MRs the tool has found and see how sensitive they are to mutations of the exiting-nomad parser.

1 Introduction

Software development and software testing go hand in hand, and software testing has become an indispensable part of the whole development process, to detect failures in the code. Especially for scientific software, that could often lead to disasters when they contain even small mistakes, good test cases can be detrimental. In the past, there have been multiple cases of huge failures, that could have been prevented by better tested software.

One prime example of that is the Mars Climate Orbiter(MCO)[Obe99]. The MCO was an integral part of NASA's Mars Surveyor Program, initiated in 1993 alongside the Mars Polar Lander (MPL). MCO had the mission of orbiting Mars, collecting weather data and serving as a relay station for data transmission to and from MPL, which was designed to land on Mars' South Pole. MCO successfully launched on the 11th of December 1998 and traveled through space for nine and a half months until it neared Mars. Unfortunately, the orbiter's signal was lost during its insertion maneuver and was never regained.

The found cause for the disaster was a failure to use the metric system in a ground software file, while another system expected these to be in the metric system. This led to the trajectory calculation software calculations being off and consequently to the failure of MCOs insertion maneuver.

Another case of a huge software failure was the Ariane 5 Flight 501[Lio+96]. The Ariane 5 was a rocket by the European Space Agency, that failed shortly after its departure. The first flight of the Ariane 5 launcher on the 4th of June 1996 did not go as planned. Shortly after starting, approximately 40 seconds into the flight and at an altitude of approximately 3700 m, the launcher deviated from its intended course, resulting in a breakup and explosion.

The cause of failure this time was the use of software from its predecessor, the Ariana 4. While there had been no problems for the Ariana 4, the Ariana 5 rocket had a stronger engine and therefore had a higher acceleration and this lead to a horizontal velocity, which is five times more rapid than for Ariane 4[Lio+96]. These high accelerations led to a larger sideways displacement, triggering the malfunction in the software, which was intended for slower accelerations.
Even though there are a multitude of ways these failures could have been avoided, better testing of the software is one of them.

One method of software testing is metamorphic testing. Unlike most traditional software testing techniques, metamorphic testing can be used to both generate test cases and verify test results. Quickly after being proposed[CCY20], it became evident, that metamorphic testing also serves as an efficient solution for addressing the oracle problem. As scientific software often includes complex algorithms and mathematical models that process massive amounts of data, it can be difficult to test by relying solely on concrete input-output relationships. Metamorphic testing has been used to complement these testing approaches for scientific software. One significant benefit is its ability to enhance test coverage by using known MRs for creating extra tests. Additionally, as scientific software often suffers from the oracle problem, because of its complexity, metamorphic testing gives us a way to bypass that.

Open challenges addressed by the thesis project. The tool by Müller et al. [Mül+22] has determined some MR candidates for some inputs they choose on some chemical materials. But it is yet to be able to generate test cases based on these MRs. Therefore, it is still not clear how sensitive the MRs are to mutations on the parser. Additionally, the data set used to determine the MRs is quite limited, so the confidence in them being correct is not very high.

For that reason, the main goal of this thesis is to tackle both of these points. To do that, we will test the MRs on calculations done by the exciting parser with different inputs and on new materials and check, how many of the original MRs still hold for them.
For the remaining MRs, we will check how well they kill the mutants, when we mutate the exciting parser. This will be done to check, how sensitive the MRs are.
The thesis is guided by the following main research questions:

RQ1 How many of the found MR candidates are likely MRs?

RQ2 How sensitive are the likely MRs to parser mutation?

Therefore, the primary contributions of this thesis are:

1. verifying the MRs and consequently the tool
2. help in testing future MR candidates for the exciting parser

2 The Context of our Study

This section is giving an overview of the state of the art and if necessary definitions and notations.

2.1 Metamorphic Testing

Metamorphic testing was first introduced as an approach to test software systems by Chen et al. [CCY20] in 1998. Their proposed method differs from traditional software testing approaches in the need of a testing oracle.
Originally, Chen et al. proposed it as a way to generate additional test cases out of already existing ones.
Normally, one would check for a test case, whether the result from the executed program matches the expected one of the oracle. If it doesn't, the test case fails. Otherwise, it is a successful or non-failing test case.
However, the availability of such a test oracle is not a given in practice.

On the contrary to the traditional method, in metamorphic testing, one doesn't need a testing oracle to do the testing. Instead of this, it looks at how the output changes when one changes the input.

Suppose you have a 20 Euro bill and want to exchange it into smaller bills. So you ask a friend how much money they have in their wallet, and they answer, that they have 30 Euros. Obviously, there is no way for you to know whether that is true or not. So you ask them, how many 10 Euro bills they have in their wallet, so you might be able to change your bill with two of these. They answer they do have five.
Clearly, we can say one of these answers has been wrong, even though we don't know how much they actually have in their wallet.

This is the main concept of metamorphic testing. Instead of looking at input-output relations, in metamorphic testing we are looking at relations between the outputs as we transform the input. For an input x and output $f(x)$, we are now looking to predict $f(t(x))$ for an input transformation $t(x)$. An example of this can be seen in the sine-function. For a specific x we might not know $sin(x)$, but we know the property of the sine-function, that $sin(\pi - x) = sin(x)$. Therefore, we can test if our original result is the same as for $sin(\pi - x)$.

2.2 Metamorphic Relations

Chen et al. has introduced metamorphic relations in "Metamorphic Testing: A New Approach for Generating Next Test Cases"[CCY20] as

> Let f be a target function or algorithm. A metamorphic relation (MR) is a necessary property of f over a sequence of two or more inputs $\langle x_1, x_2, ..., x_n \rangle$,

> where $n \geq 2$, and their corresponding outputs $\langle f(x_1), f(x_2), ..., f(x_n)\rangle$. It can be expressed as a relation R $\subseteq X^n \times Y^n$, where $\subseteq$ denotes the subset relation, and X^n and Y^n are the Cartesian products of n input and n output spaces, respectively. Following standard informal practice, we may simply write $R(x_1, x_2, ..., x_n, f(x_1), f(x_2), ..., f(x_n))$ to indicate that $\langle x_1, x_2, ..., x_n, f(x_1), f(x_2), ..., f(x_n)\rangle \epsilon$ R.

For the previous example, with f being the sine function, one MR would be $R(x, \pi - x, ..., n\pi - x, sin(x), sin(x), ..., sin(x))$, as the output for each of these inputs has to come out as the same.

2.3 Scientific Background

Scientific software can come in a lot of different ways and fields of use. It plays a critical role in nuclear industry, medicine, simulation of the climate and many more[KB14]. Due to the complexity of scientific software and the often lacking understanding of the principles behind the things, that are being calculated, it can be quite challenging to correctly test.

The main problems are, that often times the developer for the code are scientist researching the given subject, rather than programmers.
Another problem is the oracle problem, as the problems can be too complex to know what the results should be for a given input. This makes scientific software one of the main benefactors of metamorphic testing, as it is not in the need of a testing oracle.

2.3.1 NOMAD

NOMAD (Novel Material Discovery)[DS18] is a Center of Excellence with the goal to make it easier to share the discovery of improved and novel materials or unknown properties of known materials and therefore promote computational material science. It is based on the FAIR concept, which stands for findable, accessible, interoperable and reusable. In their own words, that means[DS18]

> Data are findable for any- one interested; they are stored in a way that makes them easily accessible; their representation follows accepted standards, and all specifications are open—hence data are interoperable. All of this enables the data to be used for research questions that could be different from their origi- nal purpose; hence data are repurposable.

Nomad is divides into multiple subsections:

- **The NOMAD Repository**
  The NOMAD Repository is a collection of input and output files from millions of high-quality calculations. It also incorporates data from other important computational materials databases worldwide.

- **The NOMAD Archive**
  The NOMAD Archive contains the data from the NOMAD Repository, but in a common format. The NOMAD Repository data is diverse because it is generated by different computer codes. By normalizing this data in the NOMAD Archive, it allows easier use of different NOMAD tools.
- **The NOMAD Encyclopedia**
  The NOMAD Encyclopedia functions as an access point for the NOMAD Archive. To do that, it provides a graphical UI, that allows you to navigate through all the materials.
- **The NOMAD visualization tools**
  The NOMAD visualization tools are tools, that allow the user to visualize multi-dimensional NOMAD data.
- **The NOMAD Analytics Toolkit**
  Even though the data in the NOMAD Repository and the NOMAD Archive are huge, they are still barely a fraction of the number of possible materials. Henceforth, the NOMAD Analytics Toolkit aims to gain knowledge and understanding from the already existing data.

There are over 56 supported codes with their own parser in NOMAD[1].
One of the these codes is the exciting code, which is the 7th[2] most used one in NOMAD-Lab with more than 27.3k calculations. The official website for the exciting code is saying the following about it: [3]

> exciting is a full-potential all-electron density-functional-theory package implementing linearized augmented planewave methods. It can be applied to all kinds of materials, irrespective of the atomic species involved and allows for exploring the physics of core electrons. As the name tells, exciting has a particular focus on excitations.

That means the exciting code allows the user to make calculations for all kinds of materials using linearized augmented planewave methods.
The exciting parser can read exciting input and output files and provide all the information in NOMAD's normalized format.

2.4 Automated Metamorphic Testing

Finding these metamorphic relations by hand can be quite challenging. For small datasets, it might seem easy at first, but it can get very hard for a person to spot them. As the datasets get bigger and the MRs more complex, it becomes practically impossible for them to spot all of them, or in some cases even a few.

[1]Cf. https://gitlab.mpcdf.mpg.de/nomad-lab/nomad-FAIR, 05.07.2023
[2]Cf. https://nomad-lab.eu/prod/v1/gui/search/entries, 05.07.2023
[3]https://exciting-code.org/, 14.07.2023

$$\boxed{f(3) = \tfrac{1}{4}, \quad f(6) = 1, \quad f(12) = 4, \quad f(24) = 16, \quad f(48) = 64}$$

Figure 1: For this function f with outputs for some test inputs, a possible MR is, that for $f(x) = y$, $f(2x) = 4y$.

Therefore, the same way it is done in a lot of areas, we try to minimize the human error by automating the process. Furthermore, it would save us a considerable amount of time and effort. There already exist some tools, which do exactly that. One of these has been introduced by Su et al. [Su+15] and goes by the name of KABU.
The tool finds likely MRs, as it is impossible to completely verify the correctness of them, without knowing a concrete formula for the program. They claim, that KABU can find more MPs than human developers.

Based on KABU, Müller et al. [Mül+22] developed their own tool, to achieve similar results. Just as their predecessor, they are looking to automatically find MRs, but they are fixated on scientific software, more precisely on the exiting parser for NOMAD.
They divided their approach in three phases:

1) Input Data Management phase,
2) Variable Aggregation phase
3) MR Search phase

The first phase consist of the following:

> In this phase, we generate new input data from the given sample(s), execute the parser on that data, and finally analyse the results of these parser runs.[Mül+22]

The newly generated inputs are based on the type of the variable. For Integers, the variables get possibly changed by one of the following:

1) Value increase,
2) Value decrease,
3) Value approaches ∞,
4) Value approaches $-\infty$,
5) Replace value with 0,

6) Replace value x with $-x$,

7) Use values that should not occur (e.g. a String instead of an int).

For the next phase, the Variable Aggregation phase, the tool is looking at the difference in the outputs from the original input and the new input files for each variable that is being considered by the tool, as well as the errors produced by each run. Then it saves the results in different files for each variable, because the tool is only looking for single variable metamorphic relations.

The last phase, the MR Search phase, consist, as the name implies, of searching for the MRs. Here, the previously created files with the results are getting analyzed and checked for MRs of one of following types:

1) Identity: Iff the input values of the variable are each equal to the corresponding value in the outputs, we consider this to be a strong indication for an Identity MR candidate.

2) (Strict) Monotone Increase: We sort the input files by the value of the variable under consideration. However, we keep the in- and output pairs. This means: There is no secondary sorting of the output files, instead whatever order the input files get sorted to will also be the order the output files are in. Iff we then see (strict) monotone growth across at least three files for growing input values of that variable, we consider this an indication of monotonic growth.

3) (Strict) Monotone Decrease: Similarly to the above item, iff we see (strict) monotone decreases across at least three files for growing input values of a given variable, we consider this an indication of monotonic decrease.

4) Negated inputs lead to negated outputs: Iff negating an input value of a given variable leads to the corresponding output variable being the same to the original only negated as well, we consider this to be indicative of a possible MR candidate.

5) Addition of constants: Here, we employ the Z3 theorem prover [27] to analyse whether a constant value was added across our in- and output pairs. We only consider it a MR candidate, iff the constant is added to all input values to arrive at the output values.

6) Multiplication of constants: Analogous to additions, only here we check for multiplications with constants.

[Mül+22]

Some of these are not traditional MRs, but more powerful input/output relations and would allow us to construct one out of it. For example, if one knows for a function f, that it is the identity function, they could create MRs out of that. One would be if $f(x) = y$, then $f(x + c) = y + c$ or $f(ax) = ay$.

2.4.1 Variable Detection

One Problem of the previous approaches is, that the to be considered variables still have to be chosen by experts in advance. This process is still very time-consuming. To evade this problem and to automate as much as possible, Large Language Models(LLM) have been used to automatically extract these variables[Tsi+23].
LLMs are a type of Language Model, that contain hundred of billions of parameters and are trained on a vast amount of text data[Zha+23]. These Models have the ability to understand and generate human languages and solve complex problems by predicting the likelihood of a word or phrase based on the context provided. As they are very powerful and with the rise of ChatGPT, LLMs like OpenAIs GPT-3[Bro+20] have gained a lot of attention recently.
Tsigkanos et al.[Tsi+23] proposed a method where they are used on user manuals to automatically detect variables used for metamorphic testing from them. Their results show that using the LLMs can significantly reduces the time and effort required for variable discovery.

2.5 Other Approaches

Metamorphic testing has found a home in machine learning, as Deep Neural Networks are categorized as untestable. Therefore, it is hard to create test cases for them. Metamorphic testing has created one solution for that, by being able to systematically create other tests, by using already existing ones. Consequently, it has been used by multiple people to expand their test cases.[NC19] [Xie+11]

The same way you can use Metamorphic testing to create test cases for machine learning algorithms, you can also use machine learning algorithms for finding MRs.
Kanewala et al. [KB13] proposed a method, where they create a control flow graph(CFG) for a function they want to create metamorphic relations for, and they extract a set of features from this CFG.
A CFG is a graphical representation of the possible paths a function can traverse through its execution. This representation is independent of the chosen programming language and is only dependent on the functionality of the to be considered code.
Then, a machine learning algorithm uses these features as inputs to create a predictive model.
Kanewala et al. used supervised machine learning for this.
Supervised learning is a method, where a set of labeled examples is used to learn a function. On the contrary, unsupervised learning uses unlabeled examples to learn. Common examples of use cases for unsupervised learning are clustering and anomaly detection. For example, it could be uses to detect anomalies in x-ray images.
Finally, the model is used to predict metamorphic relations using the extracted features from the CFG.

Another approach has been proposed by Zhang et al.[Zha+19], in which they introduce the tool AutoMR. This tool creates polynomial MRs with varying degrees in form of equalities and inequalities. After that, it searches for suitable parameters for these MRs with the use of particle swarm optimization(PSO). In "Particle swarm optimization: An overview"[PKB07], it is described as follows:

> In PSO a number of simple entities—the particles—are placed in the search space of some problem or function, and each evaluates the objective function at its current location. Each particle then determines its movement through the search space by combining some aspect of the history of its own current and best (best-fitness) locations with those of one or more members of the swarm, with some random perturbations. The next iteration takes place after all particles have been moved. Eventually the swarm as a whole, like a flock of birds collectively foraging for food, is likely to move close to an optimum of the fitness function.

Overall, PSO is a population-based optimization algorithm that uses the collective intelligence of particles to search for optimal solutions in a problem space through iterative refinement and adaptation.
Previously, Zhang et al.[Zha+14] proposed a similar approach, using PSO to automatically infer MRs, called MRI (Metamorphic Relation Inferrer). Although already effective, it still included redundant MRs, was limited in the types of MRs it inferred, only allow for polynomials of degrees up to two and only allowed for MRs with two inputs.
Therefore, AutoMR uses matrix singular value decomposition and constraint solving techniques to remove redundant MRs. It also allows for polynomial MRs from varying degrees and with more than two inputs.

3 Development

Our goal is to create a tool, that is able to easily check for a material, whether the given MR candidates are holding or not, for the input values we are using.

3.1 Prerequisites

The used version of the exciting parser is from 17th of February 2022, as it seems to be from around the time, as the one used in the original paper. The paper by Müller et al.[Mül+22], has been published in 2022 and presumably did not use a newer version of the exciting parser. Both the python and nomad-lab versions are the same as in the original tool1.

Name	Version
python	3.6.15
nomad-lab	0.10.4
cosmic-ray	8.3.5

Table 1: The software package used by our tool

3.2 Our Tool

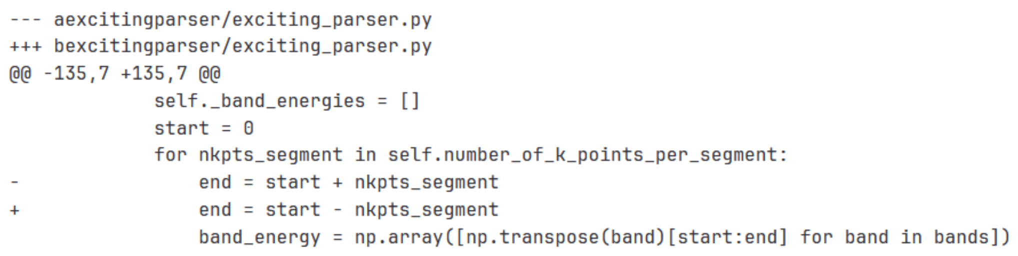
Our tool and the testing data can be found online[4] for replication purposes.
In order for our tool to help us in answering our research questions, it should be able to do the following:

1. create varying input files
2. implement all the given MR candidates
3. run the parser and use the MR candidates to test the results
4. mutate the NOMAD exciting parser

For this purpose, we divide our tool in four main sections, according to the four prior points. The first part is to create multiple input files with different inputs for each of the variables we are looking at.
Hence, we give the tool a standard input file for a material, that interests us, and it gives us multiple copies of that file, where there is a single difference in one of the for us relevant variables.
The input file has to be in the standardized form for the NOMAD-exciting parser, so the parser can run the calculations for each of the files.

[4]https://gitlab.informatik.hu-berlin.de/se/ba-klodt-daniel

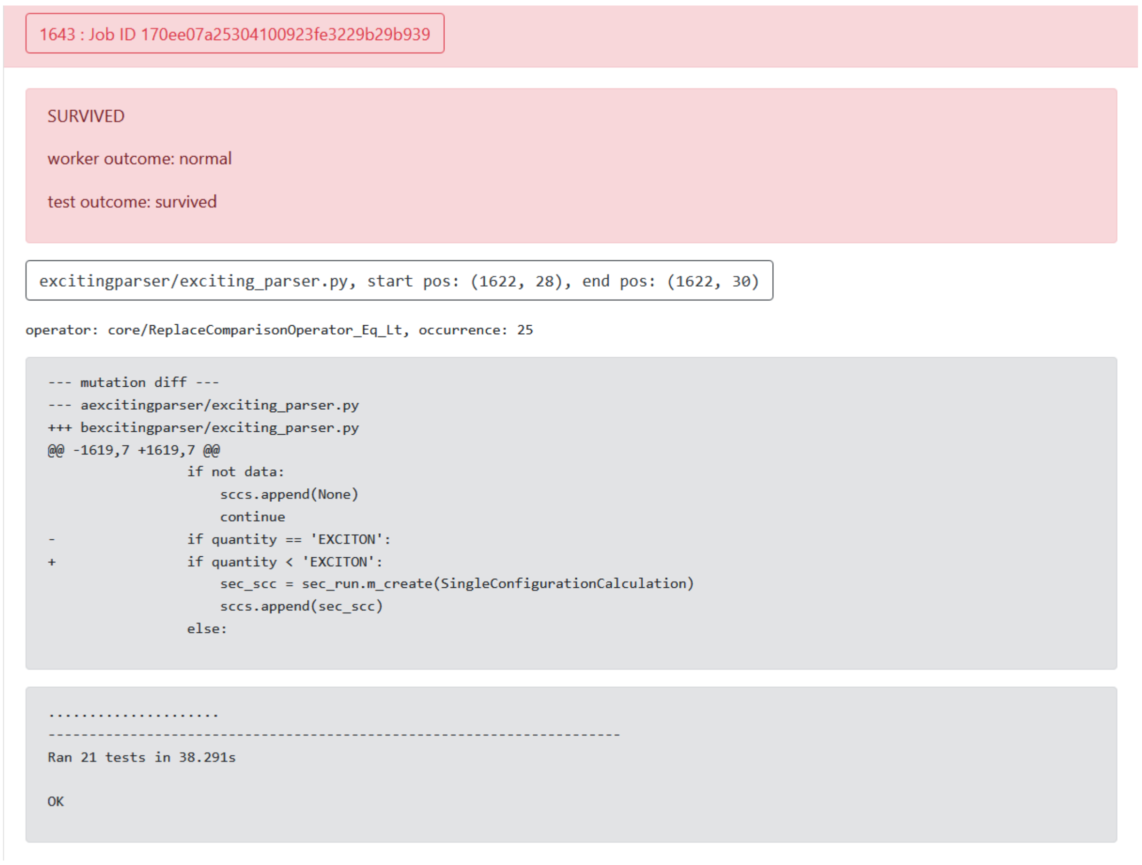
Secondly, the MR candidates determined by Müller et al.[Mül+22] need to be brought into code form, as they have been in text form so far, to let us easily test a random material with these MR candidates.
Next, we need to be able to run the exciting parser and test the previously represented MR candidates. Because metamorphic relations are looking at input-output relations, we need at least two input-output pairs to verify them. These are created by running the exciting parser on the inputs created in the first step and then comparing the results with each other. The ones, that do fail for any material, while the exciting parser has not yet been mutated, can be removed for future considerations.
That is the case, because the MRs should be a generalization and therefore hold true for all the materials. As they failed, we know, that these MR candidates were not real MRs.
We are interested in how many of these ”fake” MRs there are, to evaluate the accuracy of the tool we are looking at.
Finally, we run these tests with mutations of the exciting parser. This can be done mainly in the terminal, by using an external library for mutation testing. Our choice for this has been the cosmic-ray testing tool for python.
Cosmic-ray works by detecting possible mutations in the code, that is to be mutated, in our case the exciting parser. These mutations are always changes in exactly a single line of code at a time. Some examples for possible mutations are changing a plus signs with a minus sign or vice versa, or replacing numeric values with different ones. Figure 2 shows an example of a mutation. Here, it exchanges a plus sign with a minus sign.



```
--- aexcitingparser/exciting_parser.py
+++ bexcitingparser/exciting_parser.py
@@ -135,7 +135,7 @@
             self._band_energies = []
             start = 0
             for nkpts_segment in self.number_of_k_points_per_segment:
-                end = start + nkpts_segment
+                end = start - nkpts_segment
                 band_energy = np.array([np.transpose(band)[start:end] for band in bands])
```

Figure 2: Example for a mutation done by cosmic-ray.

The package then creates a list of all the possible mutations they found and then saves these in a database. When we then run the mutated exciting parser and test our MRs with that, cosmic-ray saves the results of the tests in exactly this database. This gives us the possibility to not run all the test in one run, as the interim results keep getting saved. These results include a Job ID, the mutation done to the code, the time it took to run all the tests and whether the mutation has been killed or not. Additionally, all the test cases, that failed or returned with an error, are shown. We write a different test case for each variable of the MRs, so one test case here is basically equivalent to an MR.
For the example in Figure 3, which is for the material CH4, we can see the tests have failed to kill the mutation and all the test cases pass. The mutation has been the exchange of the original ”==” with a ”¡” and the tests took 38.291 seconds to run.



```
1643 : Job ID 170ee07a25304100923fe3229b29b939

SURVIVED

worker outcome: normal

test outcome: survived

excitingparser/exciting_parser.py, start pos: (1622, 28), end pos: (1622, 30)

operator: core/ReplaceComparisonOperator_Eq_Lt, occurrence: 25
```

```
--- mutation diff ---
--- aexcitingparser/exciting_parser.py
+++ bexcitingparser/exciting_parser.py
@@ -1619,7 +1619,7 @@
                 if not data:
                     sccs.append(None)
                     continue
-                if quantity == 'EXCITON':
+                if quantity < 'EXCITON':
                     sec_scc = sec_run.m_create(SingleConfigurationCalculation)
                     sccs.append(sec_scc)
                 else:
```

```
.....................
----------------------------------------------------------------------
Ran 21 tests in 38.291s

OK
```

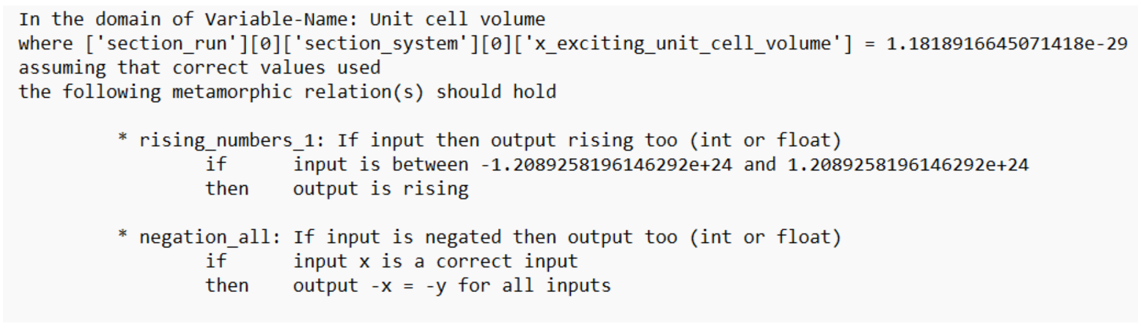
Figure 3: Example for the results for one mutation.

While running, we cosmic-ray gives the options to run the tests locally or with help of HTTP workers. Using these workers allows you to reduce the total runtime compared to running everything locally, as multiple HTTP worker can process different tests cases at the same time, save their results in the database and then move on to the next one.
The mutations allow us to see how many of the mutants have been killed and therefore give us some insights on how sensitive the remaining MRs are.
Our focus in this paper lies in MRs with the data types *int* or *float*, as there are a very small number of originally found MRs with different types (10). Furthermore, we only look at the MRs with correct values used, as we are unsure, what the meaning of the ones with wrong values used are.

3.3 Challenges

This section is going to show some of the challenges we faced, while developing our tool.

To validate the MR candidates, we had to start by bringing the MRs in a form, that allows us to that. Their original shape is in the form of a text, describing for which input and output variable the MR is valid, as seen in Figure 4. This, however, has not been fully documented, what exactly the authors meant there. We have not fully verified the meaning of each one of them, so we might have misinterpreted some of the MRs, which has been the case for MRs with wrong values used, so we removed these from our contemplation.



```
In the domain of Variable-Name: Unit cell volume
where ['section_run'][0]['section_system'][0]['x_exciting_unit_cell_volume'] = 1.1818916645071418e-29
assuming that correct values used
the following metamorphic relation(s) should hold

        * rising_numbers_1: If input then output rising too (int or float)
                if      input is between -1.2089258196146292e+24 and 1.2089258196146292e+24
                then    output is rising

        * negation_all: If input is negated then output too (int or float)
                if      input x is a correct input
                then    output -x = -y for all inputs
```

Figure 4: One MR candidate in text form.

Secondly, in the optimal case, we would like to do as many experiments as possible, to increase our confidence in the correctness of our results. The experiments however are very time-consuming, with a single run for one mutation needing somewhere from 30 up to 80 seconds. Ones where the parser last less than a second. With thousands of runs, this gets quite time-consuming.

And finally, the output files are not completely equal for all the input files. This can make it tricky to search for the exact output variable we are searching for. In some cases, the output completely failed in general and did not return the correct outputs. This also happened for Al2N2 in this paper.

4 Experiments

There are 10 different input files used for the experiments, where two of them are the same material, but still use other files. Most of them have been randomly selected from the NOMAD repository. Only one of the Gallium(III) Oxide input files (Ga4O6 original) has been selected from the ones used to determine the MR candidates. Furthermore, we decided to select a second input file for the same material at random.
Table 3 shows the all the materials used for the experiments, with the corresponding program version and the entry ID. This ID is used to identify the exact entry in the NOMAD Repository.
The experiments have been run on two different compute servers, with similar stats, as seen in Table 2, and both of them have open-SUSE Leap 15.3 as the operating system.
One of the input files, namely the one for Al2N2, has failed to produce a complete output file, where all the variables we are looking for have been missing, when run with the exciting parser. Therefore, we will be excluding that one for the evaluation.
We used the same List of Integers and Floats for all the test for computational and time reasons and to not interfere with the results.
On average, each experiment lasted approximately 8 hours, utilizing 4 HTTP workers for running the mutation test with the cosmic-ray software package. A total of 3283 mutations were carried out for each of the materials. 3283 is the number of mutations cosmic-ray was able to find for the exciting parser. We chose the automatic timeout for each test case to be 100 seconds, as a normal test case usually took us up to 70 seconds.

Model	CPU	Frequency	RAM
Dell R740	Xeon 6134	3,2GHz	756GB
Dell R740xd	Xeon 6254	3,1GHz	756GB

Table 2: Specs of the compute servers used in running the experiments. Run on open-SUSE Leap 15.3

Name	Material	Program Version	Entry ID
Al2N2	Aluminium-(III) Nitride	NITROGEN-14	vaNUWh5P4E5eFCuFyuzws8eMw2-q
BN	Boron(III) Nitride	NITROGEN-14	q5eFRvvLzpcNpXkEfp7neO8qr5ev
C2	Carbon	OXYGEN	fL0MnyQ45rsaT8nxWRddgRiFvsFD
CH4	CH4	BORON9	q9mICruqX6wRO8s1CL21tCHdbu_j
CSn4	CSn4	BORON9	zk9zyKBxg0J0Ys9MvxctqK6xxo10
Ga4O6 original	Gallium(III) Oxide	NITROGEN	znPcImrcW6PbToktDWG5MbMdo8_e
Ga4O6	Gallium(III) Oxide	CARBON	–o-KVO9NGgqP9c0yqAn_TGEniXA
GaN	GaN	BORON9	zwZ64mBu_xXLicORqfLvKmEwTs_R
HLi	HLi	OXYGEN	wxRVgMEWi-6M3ia4zNk56_harRRu
MgO	Magnesium Oxide	NITROGEN-13	6CmOFnGkf2LL6YEMgOPAXLgTNRsh

Table 3: Used materials for the mutation testing.

5 Evaluation

Of the original 160 MR candidates, we only looked at 128 ones, because the rest were for wrong values only, and we are only interested in the ones, where correct values have been used.
MRs for wrong values here mean refer to inputs with the wrong input type. An example for that would be to use 'one' as the input for a variable of the type *int*. These types of MRs did always result in an error for us, therefore we did not include them for our experiments and analysis.

5.1 Results

Of these 128 MR candidates, we found 35 to still hold, after testing them on Ga4O6 original with different input values, than the ones used previously for the same example. Next, we used the same inputs for all the other materials in Table 3. 25 of the 35 remaining ones are monotonic or strict monotonic increasing values, one is of the type identity, and the last 9 are negations, which can be seen in Table 4.
Table 5 displays, that for the mutations, excluding Al2N2, the MRs killed 21.775% of the mutations. The highest of them being BN and Ga4O6 original, each having killed 22.540% of the mutations and the lowest being HLI, with only 21.322% of the mutations killed.
To eliminate all the mutants, that have been killed, because the exciting parser does

	Total	Identity	Increasing Values	Negation	Addition	Multi-plication
original MR	128	28	36	78	9	9
remaining MR	35(27.344%)	1(3.571%)	25(69.444%)	9(11.538%)	0(0%)	0(0%)

Table 4: Number of found and remaining MR candidates for variables of type *int* or *float*.

not run anymore, we assume, that all tests, that ran for less than 5 seconds have had the mutation lead to the parser not working. The chosen value is 5 seconds, as it takes less than a second to run the tests with a failed parser and more than 10 seconds, but mostly more than 20 seconds, to run a single working test.
Furthermore, we are interested in the mutants killed by the MRs, as in the ones, where the output stays an Integer or Float, but do not fulfill the conditions of the MR. Therefore, we also remove the ones, where the output we are looking at has become a NoneType.
Table 6 shows how many of the killed mutations have been killed due to the MRs themselves and how many were killed because the parser failed to work.
Of the combined 6434 killed mutations, a total of 650 have been killed by the MR, which means, that the test took more than 5 seconds and did not time out. That is a total of 10.103% of all the killed mutations. When we further remove all tests, where the mutation has only been killed because of a NoneType output, we are left with 386 remaining mutations killed, which is 5.999% of the killed mutations. Lastly, 11 runs did time out(0.171%).
Finally, Table 7 shows how many of the mutations our MRs have been able to detect of all the runs, when we exclude the ones, that took less than 5 seconds to run. The MRs did detect 2.735% of all the mutations and if we exclude the ones detected by the output being NoneTypes, we are left with 1.624%.

5.2 Analysis and Discussion

We asked in RQ1 how many of the found MR candidates are likely MRs. This can be answered pretty straight forward:

Of the 128 MR candidates we looked at, 93, which is 72.656%, have been found to not be real MRs. For the remaining 35(27.344%) we assume them to indeed be true MRs, as they passed every test for us. This means, we estimate, around $\frac{1}{4}$ of the MR candidates the tool found are indeed MRs, and therefore are likely MRs.

For our second question, RQ2, we asked: How sensitive are the likely MRs to parser mutation?

Material	Total Runs	surviving mutants	killed mutants	percentage killed
Al2N2	3283	0	3283	100%
BN	3283	2543	740	22.540%
C2	3283	2580	703	21.413%
CH4	3283	2577	706	21.505%
CSn4	3283	2581	702	21.383%
Ga4O6 original	3283	2543	740	22.540%
Ga4O6	3283	2581	702	21.383%
GaN	3283	2545	738	22.479%
HLi	3283	2583	700	21.322%
MgO	3283	2580	703	21.413%
Total(no Al2N2)	29547	23113	6434	21.775%

Table 5: Number of killed mutants for each input file.

In order to discuss the sensibility of the MRs, we first have to say, what that means for our experiments. We looked at the ability of the MRs to kill mutations in the code of the NOMAD exciting parser. Optimally, we would like them to be able to kill all of them, which would also make them very sensitive to parser mutation. On the contrary, not being able to detect any mutations would mean the mutations are very insensitive to parser mutation.

As the goal of creating the MRs is, to help in testing code and detect faults in it, completely insensitive MRs lose their value, even if they are correct, as they will not help us in detecting any faults in the code.

But not all mutations are the same. A lot of them directly lead to the parser not working at all. Even though cosmic-ray shows these cases as the MRs having killed the mutation, in reality, these runs would have failed for any test case. Consequently, these are mutations, that we did remove from consideration in our experiments.

The next group of mutations we separated are ones, where the output type of the variables we are looking at are NoneTypes. Despite the MRs detecting mutations, which are leading to NoneTypes in the output, as they need the outputs to be of the correct data type, we could also always make sure, the output type is correct, to remove these faults.

Therefore, even though there are MRs, that are not able to detect these types of mutations, we would like the MRs to kill mutations based on their intended purpose, looking at the numeric values of the outputs.

Material	killed mutants	killed by MR	no NoneType	timeout(100s)
Al2N2	3283	0	0	0
BN	740	166	126	5
C2	703	18	0	0
CH4	706	24	0	0
CSn4	702	32	1	0
Ga4O6 original	740	183	143	4
Ga4O6	702	29	0	0
GaN	738	153	116	2
HLi	700	27	0	0
MgO	703	18	0	0
Total(no Al2N2)	6434	650(10.102%)	386(5.999%)	11(0.171%)

Table 6: Number of killed mutants by the way they have been killed.

The experiments show the likely MRs to be rather insensitive. Even though 21.775% of the mutation could be killed, most of these were not due to our MRs themselves, but rather the exciting parser failing in general.
They have only been able to kill 1.624% of the mutations, when we exclude all runs, where it took less than 5 seconds for the test and only count the ones, where the resulting variables we are looking at are not NoneTypes. This is a rather low number and indicates, that our MRs are not very sensitive at all.

The MRs that passed the initial checks show low sensitivity to parser mutation. While many mutations have been eliminated, the majority of mutants were killed due to complete parser failure rather than the MRs themselves. However, the MRs were successful in detecting and eliminating a small portion (1.624%) of the mutations in our experiments.

5.3 Threats to Validity and Future Work

For our work, we used the same mutations for all the different materials, as the cosmic-ray package did only find these mutations. It might be purposeful to also increase the number of mutations for each of the materials instead of looking at more materials, as the behavior of the parser seems to be fairly consistent for most of the different materials.
Another major threat to validity is, that we did not communicate much with the original authors, who found the MR candidates we examined. Because the original MR

Material	total runs(no short ones)	killed by MR	no NoneType
BN	2709	166	126
C2	2598	18	0
CH4	2601	24	0
CSn4	2613	32	1
Ga4O6 original	2726	183	143
Ga4O6	2610	29	0
GaN	2698	153	116
HLi	2610	27	0
MgO	2598	18	0
Total	23763	650(2.735%)	386(1.624%)

Table 7: Percentages of mutations killed by the parser for all runs, that needed more than 5 seconds.

candidates were captured as texts, we might have misunderstood the meaning of some of them. For this exact reason, we also did exclude some of the MR candidates, namely the one for wrong values, as the meaning of these MRs seemed unclear.
Furthermore, we used a relatively short list of Integers and Floats for our experiments. This helped us reduce the time of the experiments, but incidentally also reduces the chance of the MRs to find mutations.
Additionally, our mutations did only cover a single line of code at a time. For a program as big as the exciting parser, this may lead to the exciting parser either completely failing, when an essential line has been modified, or not doing anything at all. It is highly unlikely for a change in one line of code to only modify the numeric value of a single output.
Lastly, the fact that the MRs basically only found mutations for 3 of the 9 input files, while they found more than a hundred for each of these 3 is something we are not able to explain yet and won't be able to further investigate for time constraints. One possible reason could be a shared property between these input files, that we are not aware of. As we used two different input files for Ga4O6, where we have been able to kill multiple mutations for one of them and not a single one for the other, it is unlikely to be a property of the material itself.

Future work could address some of these problems, mainly the lack of mutations of the code for the exciting parser, as well as the type of mutations, as we always only mutated a single line of code.
Our work also suggests the main improvements to be made are with the MRs themselves. They seem to be too simple and insensitive for code as big as the one for the exciting parser, and therefore, it would be sensible to look for MRs, which circumvent the problem by increasing their complexity.

One way to do that, would be to look for MRs with multiple inputs and/or outputs or use more complex functions as a base for the MRs. These could come in a multitude of different forms, such as polynomial functions of varying degrees.
Finally, further inspection of the killed mutations could be helpful in explaining, why the MRs could only detect mutation for some of the materials, while they have been completely unable to find ones for the other ones.

6 Conclusion

In this paper, we looked at some MR candidates determined in previous work, that had yet to be fully verified or evaluated. Therefore, we checked the validity of the MR candidates for multiple different materials and some new input values, and also looked at how sensitive they are to mutation of the used NOMAD exciting parser. By doing that, we also created a tool, that allows us to easier test future MRs for the same parser.
We showed, that those found so far do not generalize well and are quite insensitive to mutations of single lines of code in the parser, as they could not kill a single mutation for multiple of the materials.
This matches our original assumption, that the metamorphic relations are not powerful enough to confidently detect small changes for big programs.

References

[Bro+20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[CCY20] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. “Metamorphic testing: a new approach for generating next test cases”. In: *arXiv preprint arXiv:2002.12543* (2020).

[DS18] Claudia Draxl and Matthias Scheffler. “NOMAD: The FAIR concept for big data-driven materials science”. In: *Mrs Bulletin* 43.9 (2018), pp. 676–682.

[KB13] Upulee Kanewala and James 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)*. IEEE. 2013, pp. 1–10.

[KB14] Upulee Kanewala and James M Bieman. “Testing scientific software: A systematic literature review”. In: *Information and software technology* 56.10 (2014), pp. 1219–1232.

[Lio+96] Jacques-Louis Lions, Lennart Luebeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, et al. *Ariane 5 flight 501 failure report by the inquiry board*. 1996.

[Mül+22] Sebastian Müller, Valentin Gogoll, Anh Duc Vu, Timo Kehrer, and Lars Grunske. “Automatically finding Metamorphic Relations in Computational Material Science Parsers”. In: *2022 IEEE 18th International Conference on e-Science (e-Science)*. IEEE. 2022, pp. 521–528.

[NC19] Shin Nakajima and Tsong Yueh Chen. “Generating biased dataset for metamorphic testing of machine learning programs”. In: *Testing Software and Systems: 31st IFIP WG 6.1 International Conference, ICTSS 2019, Paris, France, October 15–17, 2019, Proceedings 31*. Springer. 2019, pp. 56–64.

[Obe99] James Oberg. “Why the Mars probe went off course [accident investigation]”. In: *IEEE Spectrum* 36.12 (1999), pp. 34–39.

[PKB07] Riccardo Poli, James Kennedy, and Tim Blackwell. “Particle swarm optimization: An overview”. In: *Swarm intelligence* 1 (2007), pp. 33–57.

[Su+15] Fang-Hsiang Su, Jonathan Bell, Christian Murphy, and Gail Kaiser. “Dynamic inference of likely metamorphic properties to support differential testing”. In: *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*. IEEE. 2015, pp. 55–59.

[Tsi+23] Christos Tsigkanos, Pooja Rani, Sebastian Müller, and Timo Kehrer. “Variable Discovery with Large Language Models for Metamorphic Testing of Scientific Software”. In: *International Conference on Computational Science*. Springer. 2023, pp. 321–335.

[Xie+11] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, et al. “Testing and validating machine learning classifiers by metamorphic testing”. In: *Journal of Systems and Software* 84.4 (2011), pp. 544–558.

[Zha+19] Bo Zhang, Hongyu Zhang, Junjie Chen, Dan Hao, and Pablo Moscato. “Automatic discovery and cleansing of numerical metamorphic relations”. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2019, pp. 235–245.

[Zha+14] Jie Zhang, Junjie Chen, Dan Hao, Yingfei Xiong, Bing Xie, et al. “Search-based inference of polynomial metamorphic relations”. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 701–712.

[Zha+23] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, et al. “A survey of large language models”. In: *arXiv preprint arXiv:2303.18223* (2023).

Appendix

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird. Berlin, den

July 22, 2023 ....