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

Metamorphes-Testen in computergestützten Daten Analysis Workflows in den Materialwissenschaften

Bachelorarbeit

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

eingereicht von: Valentin Gogoll
geboren am: 27.04.1999
geboren in: Pritzwalk

Gutachter/innen: Prof. Dr. Lars Grunske
Prof. Dr. Timo Kehrer

eingereicht am: ..... verteidigt am: .....

Inhaltsverzeichnis

Abstract

Ein wichtiger Bestandteil der Softwareentwicklung ist das Testen, jedoch ist das Testen wissenschaftlicher Software teilweise schwieriger als das Testen von Programmen, bei welchen die Ausgabe im Vorhinein bekannt ist. Metamorphes Testen ist ein Ansatz, bei welchem mit Hilfe von metamorphen Relationen gearbeitet wird. In unserer Arbeit untersuchen wir, ob es möglich ist automatisiert auf dem `exciting`-Parser nach metamorphen Relationen zu suchen. Ebenso betrachten wir, welche Relationen gefunden werden können. Der `exciting`-Parser gehört zu `exciting`, was ein NOMAD-Code ist. NOMAD verwaltet Simulationen aus den Materialwissenschaften. Um die Frage dieser Arbeit zu klären, haben wir ein eigenes Werkzeug zur Suche von metamorphen Relationen implementiert. Letztendlich konnten wir zeigen, dass unser Werkzeug metamorphe Relationen auf dem `exciting`-Parser finden kann. Ein Teil dieser Relationen ließ sich auf unseren Daten als allgemeingültig verifizieren. Ob der `exciting`-Parser damit tatsächlich getestet werden kann und ob unser Werkzeug auf die Parser der weiteren NOMAD-Codes angewendet werden kann, sind Themen für weitere Arbeiten.

1 Einleitung

Software ist in den letzten Jahrzehnten immer mehr zum Bestandteil unseres Alltags geworden. Dass dabei nicht immer alles funktioniert, ist bekannt. Immer wieder treten Fehler in Apps auf oder Programme funktionieren nicht, wie sie sollen. Meist sind es Fehler, über die man hinwegsehen kann. Jedoch gab es in der Vergangenheit bereits Softwarefehler, die deutlich schlimmere Folgen nach sich zogen, als ein abstürzender PC. Der Verlust der ersten Ariane-5-Rakete wäre nur ein nennenswertes Beispiel. Im Jahre 1996 sprengte sich diese Rakete kurz nach dem Start aufgrund eines Softwarefehlers selbst, was zu einem Schaden von rund 370 Millionen US-Dollar führte. [Mar97] Um solche Fehler zu verhindern, ist das Testen ein wichtiger Bestandteil der Softwareentwicklung. Jedoch ist das Testen sehr zeit- und kostenintensiv. Um dem entgegenzuwirken, ist das Ziel, das Testen von Software soweit wie möglich zu automatisieren.

Auch in der Forschung wird auf Softwareprogramme zurückgegriffen, ebenso bei den Materialwissenschaften. In diesem Gebiet wird nach neuen Materialien für bestimmte Anwendungen gesucht und bekannte Materialien weiter erforscht. Um die Forschung auf diesem Gebiet zu verbessern, wurde NOMAD (Novel Materials Discovery) gegründet. NOMAD unterstützt aktuell bereits über 40 Programme (sogenannte DFT Codes) [1], mit denen verschiedene Berechnungen bzw. Simulationen zu Materialien durchgeführt werden können. Die bei den Berechnungen erzeugten Daten können dann in eine weltweit verfügbare Datenbank, das NOMAD-Archiv, eingespielt werden. Dazu müssen die generierten Daten vorher übersetzt und bereinigt werden, sodass diese in einem einheitlichen Datenformat vorliegen. [vgl. Cla18]

[1] `https://nomad-lab.eu/prod/rae/gui/search`

Um alle Daten in ein einheitliches Format zu übersetzen, werden sogenannte Parser verwendet. Zur Veranschaulichung eines Parser haben wir in Abbildung 1 grob den Ablauf dargestellt, wie die generierten Daten übersetzt werden.

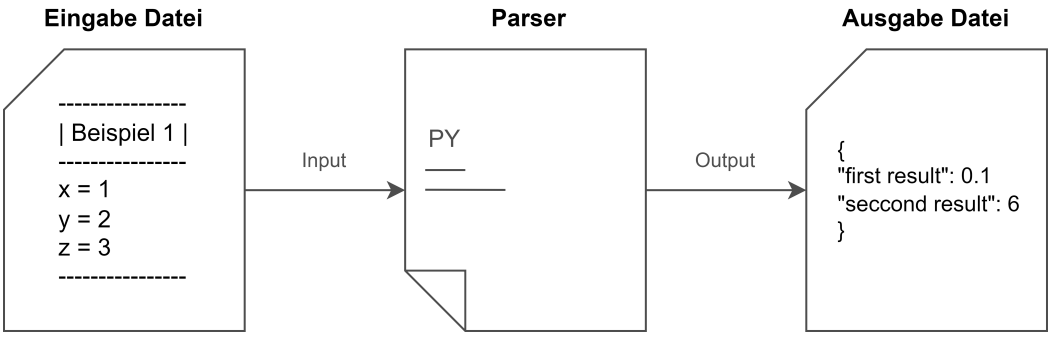

Abbildung 1: Beispiel-Parser - Einsatzbeispiel für einen Parser. Der dargestellte Beispielparser übersetzt die Eingabe Datei auf der linken Seite und liefert die Ausgabe Datei auf der rechten Seite als Ergebnis.

Im Vergleich zu dem Beispiel in Abbildung 1 sind die Parser der NOAMD-Codes deutlich umfangreicher und verarbeiten mehr Informationen. Jedoch basieren auch die Eingaben einiger NOMAD-Parser auf einem ähnlichen Textformat wie in Abbildung 1 zu sehen, ebenso wird die Ausgabe im JSON-Format zurückgegeben.

Zu jedem von NOMAD unterstütztem Code gibt es einen dazugehörigen Parser – eine mögliche Fehlerquelle. Zwar werden die Daten, welche in das NOMAD-Archiv eingespielt werden, zuvor auf ihre Plausibilität geprüft, sodass grobe Fehler auffallen und nicht das Archiv „verunreinigen“ können. Wenn jedoch einer dieser Parser Daten falsch verarbeitet und dieser Fehler nicht offensichtlich erkennbar ist, könnten fehlerhafte Informationen in die Datenbank gelangen. Wenn diese fehlerhaften Informationen für weitere Berechnungen verwendet werden, kann dies die Berechnungen verfälschen und zu ungenauen oder falschen Ergebnissen führen.

Um solche Probleme zu verhindern, sollten diese Parser systematisch getestet werden. Allgemein kann nicht jedes Programm auf die gleiche Art und Weise getestet werden. Vor allem wissenschaftliche Software stellt ein Problem beim Testen dar, denn meist ist die korrekte Ausgabe des Programms im Voraus nicht bekannt, weshalb auf trivialem Weg nicht entschieden werden kann, ob sich das Programm bei der Ausführung eines Testfalls korrekt oder fehlerhaft verhält. [vgl. Dan09] Ein weiterer Faktor, der das Testen erschwert, sind großen Datenmengen, die in der Forschung verarbeitet werden. Das Problem der NOMAD-Parser ist, dass meist Daten in der Größenordnung von mehreren Mega- bis Gigabytes durch die Parser verarbeitet werden. [vgl. Dan09]

Metamorphes Testen (MT) ermöglicht es, auch Programme ohne ein vorhandenes Testorakel zu testen. MT basiert auf Relationen zwischen den Ein- und Ausgaben des zu testenden Programms. Mit Hilfe von Transformationen können so, basierend auf

der Ein- und Ausgabe eines Testfalls, neue Testfälle entworfen werden. [vgl. Ser16] Dabei stellt das Finden und Bewerten der Relationen eine Herausforderung dar, denn die Auswahl der Relationen und des Starttestfalls entscheiden maßgebend über die Qualität der daraus resultierenden Testsuite.

In unserer Arbeit möchten wir herausfinden, ob es möglich ist, MT auf den Parsern der von NOMAD unterstützten Codes anzuwenden. Jedoch werden wir unseren Ansatz nur auf dem `exciting`-Parser anwenden und testen. In der Arbeit von Upulee Kanewala und James M. Bieman [Upu13a] wird MT mit dem Testverfahren „cross-validation“ verglichen und stellt sich als effizienter heraus. „cross-validation“ ist ein weiteres Testverfahren für Programme ohne Testorakel. Die Idee hinter „cross-validation“ ist es, einen Datensatz zu unterteilen. Ein Teil des Datensatzes wird zum trainieren der Methode verwendet, der andere zum Testen der Methode. Mit „cross-validation“ lässt sich eine Aussage darüber treffen, wie gut ein Vorhersagemodell mit zuvor unbekannten Daten umgehen kann. [vgl. Upu13a]
Jedoch gibt es bisher kein allgemeines MT Werkzeug, dass sich direkt auf die NOMAD Parser anwenden lässt. Wir wollen einen MT Prototypen bauen, mit dem wir die `exciting`-Parser testen können. Damit wollen wir einen Teil dieser Lücke schließen.

Unser Werkzeug soll metamorphe Relationen (MR) finden, auf deren Basis in einem weiteren Schritt Testfälle entworfen/generiert werden können, mit denen sich der Parser testen lässt. Dennoch können diese Testfälle keine Fehler finden, welche möglicherweise bereits im Parser vorhanden sind. Die Testfälle können aber Regressionen in zukünftigen Versionen finden. Aus einer gefundenen Regression lässt sich jedoch nicht direkt ableiten, ob es sich um einen Fehler handelt, welcher in der neuen Version vorhanden ist oder ob die neue Version einen bestehenden Fehler beseitigt. Um dies zu entscheiden, muss die gefundene Regression manuell untersucht werden.

Konkret lassen sich die Forschungsfragen, welche wir im Laufe dieser Arbeit beantworten wollen, wie folgt zusammenfassen:

RQ1: Ist es machbar und darüber hinaus sinnvoll ein Programm zum automatisierten Suchen von MR auf dem `exciting`-Parser zu entwickeln?

RQ2: Kann dieses Programm MR finden und wenn ja, wie hoch ist die Qualität und Quantität dieser MR?

Im ersten Teil der Arbeit werden wir alle nötigen Fachbegriffe und Methoden erläutern, welche für diese Arbeit nötig sind (Abschnitt 2). Auch werden wir einen Einblick in den aktuellen Stand der Technik geben (Abschnitt 3).
Im darauf folgenden Kapitel (Abschnitt 4) werden wir den Aufbau unseres Programms und die Entwicklung dessen beschreiben. Auf Herausforderungen, welche während der Entwicklung aufgetreten sind, werden wir in Unterabschnitt 4.3 eingehen.
Die Auswertung unseres Programms wird in Abschnitt 5 stattfinden. Dazu werden wir mehrere `exciting`-Berechnungen als Grundlage für das von uns entwickelte Werkzeug

verwenden und untersuchen, wie viele MR sich finden lassen. Auch werden wir die gefundenen MR an sich auswerten.

2 Hintergrund

In diesem Kapitel werden die grundlegenden Begriffe erklärt, welche für diese Arbeit nötig sind. Ebenso werden wir die in der Arbeit verwendeten Programms erklären.

2.1 Metamorphes Testen

Das Metamorphe Testverfahren wurde erstmals 1998 von T. Y. Chen et. al. in seiner Arbeit „Metamorphic testing: A new approach for generating next test cases“ beschrieben. Im Gegensatz zu unserer Arbeit, stand in der Arbeit von Chen et. al. das Erzeugen neuer Testfälle im Vordergrund. Das Generieren neuer Testfälle basiert dabei auf den Eingabe-Ausgabe-Paaren eines erfolgreichen Basistestfalls. Das Ziel ist es, so weitere Fehler zu finden, welche sonst möglicherweise unentdeckt bleiben würden. [vgl. T. 98]

Wenn ein Testfall erfolgreich ausgeführt wurde, sagt dies nicht unbedingt aus, dass in dem zu testenden Programm keine weiteren Fehler vorhanden sind. Dies hat bereits Dijkstra in seiner Arbeit [Eds72] beschrieben:

> [...] [P]rogram testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

Da Testfälle nicht die Abwesenheit von Fehler beweisen können, ist es wichtig, möglichst gründlich zu testen bzw. eine möglichst hohe Testabdeckung zu erreichen, um die Qualität des Programms sicherzustellen. An dieser Stelle setzt MT an, indem aus bereits positiv ausgeführten Testfällen weitere Testfälle erzeugt werden. So kann eine höhere Testabdeckung erreicht werden. Ein weiterer Vorteil dieses Verfahrens ist es, dass MT auch ohne Test-Orakel angewendet werden kann. Somit muss nicht genau bekannt sein, wie die korrekte Ausgabe des zu testenden Programms aussieht. Trotzdem wird teilweise domain-spezifisches Wissen benötigt. Ein Beispiel dafür wäre, metamorphe Relationen zu finden und zu bewerten. Auch für die Auswahl, möglichst effizienter MR wird domain-spezifisches Wissen benötigt. [vgl. T. 98]

Mit Hilfe von MT lässt sich das Orakel-Problem umgehen. Dies ermöglicht es, auch solche Software zu testen, von der im Voraus nicht bekannt ist, ob ein Ergebnis korrekt ist oder nicht. Beispielsweise könnte dies auf folgende Programmgruppen zutreffen: Compiler, Simulatoren oder Suchmaschinen. [vgl. Ser19]

Auch bei wissenschaftlicher Software ist nicht immer bekannt, ob ein berechnetes Ergebnis korrekt ist. Teilweise ist es in manchen Situationen nicht möglich, das Ergebnis nachzuberechnen. Jedoch kann zum Testen solcher Programme MT verwendet werden.

2.2 Metamorphe Relationen

Um das Metamorphe Testverfahren verwenden zu können, werden Metamorphe Relationen benötigt. Diese beschreiben den Zusammenhang zwischen den Ein- und Ausgabepaaren. In der Arbeit von T. Y. Chen werden Metamorphe Relationen wie folgt definiert [T. 98]:

> Let f be a target function or algorithm. A metamorphic relation is a necessary property1 of f over a sequence of two or more inputs $\langle x_1, x_2, \ldots, x_n \rangle$, where $n \geqslant 2$ and their corresponding outputs $\langle f(x_1), f(x_2), \ldots, f(x_n) \rangle$. It can be expressed as a relation $\mathcal{R} \subseteq X^n \times Y^n$ 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 $\mathcal{R}(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n))$ to indicate that $\langle x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n) \rangle \in \mathcal{R}$

Ins Deutsche übersetzt sagt die Definition folgendes aus [vgl. T. 98]:

Sei f eine Funktion oder ein Algorithmus. Eine Metamorphe Relation ist eine notwendige Eigenschaft von f über einer Sequenz von zwei oder mehr Eingaben $\langle x_1, x_2, \ldots, x_n \rangle$, mit $n \leqslant 2$, und den dazugehörigen Ausgaben $\langle f(x_1), f(x_2), \ldots, f(x_n) \rangle$. (Dabei ist eine notwendige Eigenschaft eine Bedingung, welche sich logisch aus dem Algorithmus oder der Funktion ableiten lässt.) Dies kann als Relation $\mathcal{R} \subseteq X^n \times Y^n$ ausgedrückt werden, wobei $\subseteq$ die Teilmengen-Relation bezeichnet und X^n und Y^n die kartesischen Produkte der n Eingabe- bzw. Ausgaberäume sind.

Ein Beispiel für eine solche metamorphe Relation wäre Folgendes: Angenommen der obige Beispielparser aus Abbildung 1 würde die Eingabe-Daten bekommen, wie in Abbildung 2 dargestellt. Dann würde der Parser die in Abbildung 3 dargestellten Dateien als Ausgabe zurückgeben.

Beispiel 1	Beispiel 2	Beispiel 3
x = 2 y = 3 z = 4	x = 3 y = 4 z = 5	x = 4 y = 5 z = 6
a: Beispiel-Eingabe 1	b: Beispiel-Eingabe 2	c: Beispiel-Eingabe 3

Abbildung 2: **Beispiel-Eingaben:** Dargestellt sind drei Eingabe-Dateien, welche fiktiv durch unseren Beispiel-Parser übersetzt werden sollen. Beim Vergleich mit den Ausgabe-Dateien, lassen sich MR finden.

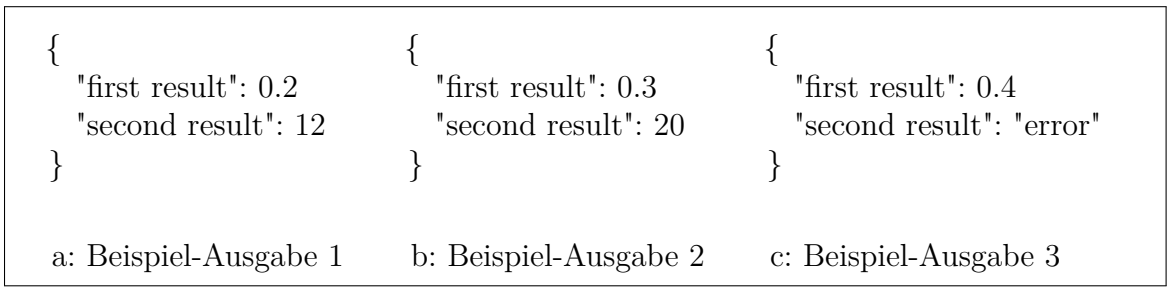

Abbildung 3: **Beispiel-Ausgaben:** Dargestellt sind die Ausgabe-Dateien, welche unser Beispiel-Parser< (Abbildung 1)) aus den in Abbildung 2 dargestellten Beispiel-Eingaben erzeugt.

Betrachtet man nun die Ein- und Ausgabe-Dateien, so fällt Folgendes auf:
Der Wert für „first result“ verringert sich beim Übersetzen jedes Mal auf ein Zehntel der Eingabe von x. Der Wert für „second result“ steigt, wenn auch die Eingabewerte für y und z steigen. Ebenso ist erkennbar, dass es in der letzten Ausgabe zu einem Fehler beim „second result“ kommt.

Auch wenn wir nicht genau wissen, wie die Umrechnung im Hintergrund aussieht, können wir daraus Relationen ableiten, mit welchen sich weitere Berechnungen des Parser zum Teil vorhersagen lassen. Diese MR haben wir in Abbildung 4 aufgelistet.

Im Bereich des *Beispiel-Parsers*
sollten folgende metamorphe Relation(en) gelten

- MR1: Umrechnung von x:

  Wenn x eine gültige Eingabe ist,

  dann ergibt sich der Wert „first result“ in der Ausgabe aus $x \cdot 0.1$.

- MR2: Relation zwischen y und z:

  Wenn y und z eine gültige Eingabe sind und ihre Werte steigen,

  dann steigt auch der Wert für „second result“.

- MR3: Wertebereich von y und z:

  Wenn für y der Wert 5 und für z der Wert 6 übergeben wird<,

  dann ergibt sich der Wert „second result“ in der Ausgabe aus der Zeichenkette „error“.

Abbildung 4: MR des Beispielparser welche sich aufgrund der in Abbildung 2 und Abbildung 3 dargestellten Ein- und Ausgabe-Dateien ableiten lassen.

Die Notation der MR orientiert sich dabei an einer Vorlage, welche in „A Template–Based Approach to Describing Metamorphic Relations“ beschrieben wurde [Ser17]. Konkret wird dort folgende Vorlage zum Beschreiben von MR präsentiert:

> **In the domain of** <application domain>
> [**where**<context definition>]
> [**assuming that**<constraints>]
> **the following metamorphic relation(s) should hold**
>
> - <metamorphic relation name_1>:
>
> **if** <relation among inputs/outputs>
>
> **then** <relation among inputs/outputs>
>
> ...
> - <metamorphic relation name_n>:
>
> **if** <relation among inputs/outputs>
>
> **then** <relation among inputs/outputs>

2.3 NOMAD

Der Name NOMAD steht für „Novel Materials Discovery“ und ist ein CoE (Center of Excellence), also ein Zusammenschluss von mehreren Experten, unter anderem aus den Materialwissenschaften. Das Ziel von NOMAD ist es, die Suche nach neuen Materialien zu verbessern, indem bereits vorhandene Daten und Berechnungen zu verschiedenen Materialien öffentlich zugänglich gemacht werden. Diese können von anderen Wissenschaftlern wiederverwendet werden, damit Berechnungen nicht erneut durchgeführt werden müssen, um Zeit und Ressourcen zu sparen.

FAIR — Findable, Accessible, Interoperable, Reusable:

Bei NOMAD verläuft der Umgang mit den Daten nach dem FAIR Prinzip, welches Folgendes aussagt [Cla18]:

> Data are Findable for anyone interested; they are stored in a way that they are easily Accessible; their representation follows accepted standards [13], and all specifications are open – hence data are Interoperable [14]. All this enables that data can be used for research questions that can be different from the purpose they have been created for; hence data are Repurposable.

Mit anderen Worten, die Daten, welche verwaltet werden, sollen von jedem gefunden werden können, der Interesse an diesen Daten hat. Auch sollen die Daten leicht zugänglich sein. Die Struktur der Daten soll sich an akzeptierte Standards halten. Spezifikationen, welche die Daten betreffen, sollen öffentlich bzw. nachvollziehbar sein. Der letzte Punkt bezieht sich auf die Wiederverwendbarkeit. Die Daten sollen so gehandhabt werden, dass andere diese auch für ihre Zwecke wiederverwenden können. [vgl. Mar16]

NOMAD lässt sich dabei in die folgenden fünf Teile aufspalten [vgl. Cla18]:

NOMAD Repository: Dieses Repository enthält Rohdaten, unter anderem die Ein- und Ausgabedaten verschiedener Berechnungen, welche bereits zu bestimmten Daten und Materialien durchgeführt wurden.

NOMAD Archive: Das NOMAD Archiv besteht aus den im NOMAD – Repository liegenden Rohdaten, welche durch verschiedene Parser in ein Code-Unabhängiges Format übersetzt wurden. Die Übersetzung der Daten beinhaltet unter anderem eine Normalisierung, welche die Möglichkeit bietet, die Daten verschiedener Berechnungen (Codes) vergleichen zu können.

NOMAD Visualistion Tools: Diese Werkzeuge ermöglichen den Forschern, Daten bequem visuell zu analysieren.

NOMAD Encyclopedia: Die NOMAD Encyclopedia ist eine nutzerfreundliche, grafische Oberfläche, über welche alle berechneten Eigenschaften und weitere Daten zu den Materialien eingesehen werden können, welche im NOMAD Archive hinterlegt sind.

NOMAD Analytic Tool: Um beispielsweise die Suche nach Materialien mit bestimmten Eigenschaften zu verbessern, entwickelt NOMAD derzeit das Analytic Tool. Dieses Werkzeug durchsucht die Daten im NOMAD-Archiv. Eine konkrete Funktion soll unter anderem die Suche nach besseren Materialien für die „heterogene Katalyse“ sein.

Die einzelnen Programme, mit welchen verschiedene Berechnungen zu Materialien durchgeführt werden können, heißen Codes. Derzeit unterstützt NOMAD über 40 Codes, dabei ist VASP die meist verwendete Methode. All diese Codes besitzen einen eigenen Parser, um die erzeugten Daten zu normalisieren und in ein Code-unabhängiges Format zu übersetzen. Bisher gibt es keine Informationen zu strukturierten Tests dieser Parser, daher wollen wir uns exemplarisch den `exciting`-Parser anschauen und überprüfen, ob sich dieser Parser metamorph testen lässt. Der Grund dafür ist, dass die Betrachtung weiterer Parser den Rahmen der Arbeit überschreiten würde. Ob sich auch die anderen Parser metamorph testen lassen, wäre ein Thema für eine spätere Arbeit.

2.4 Der exciting-Code

Der `exciting`-Code ist derzeit auf Platz 8[2] der meistverwendeten Codes bei NOMAD. Laut der offiziellen Internetseite bietet der Code folgende Berechnungen [exc]:

> `exciting` is a full-potential all-electron density-functional-theory package implementing the families of linearized augmented planewave methods. It

[2] `https://nomad-lab.eu/prod/v1/gui/search/entries` abgerufen am 09.10.2021

> can be applied to all kinds of materials, irrespective of the atomic species involved, and also allows for exploring the physics of core electrons. A particular focus are excited states within many-body perturbation theory.

Mit andern Worten, `exciting` ist ein „Allelektronendichtefunktionaltheorie-Paket“ und ermöglicht Berechnungen aus der Familie der linearisierten erweiterten Planewave-Methoden, wobei alle Potentiale der Elektronen berechnet werden. Anwenden lässt sich `exciting` auf alle Materialien, unabhängig der Atom-Art. Ebenso wird die Erforschung der Physik von Kernelektronen ermöglicht. Dabei spielt die Vielteilchen-Störungstheorie eine wichtige Rolle.
Im Gegensatz zum Parser besitzt der `exciting`-Code eine automatisierte Testsuite. [vgl. exc]
Derzeit wird der `exciting`-Code von der Arbeitsgruppe „Festkörpertheorie“ des Physikalischen Instituts der Humboldt-Universität zu Berlin, unter der Leitung von Prof. Claudia Draxl, weiterentwickelt. `exciting` wurde in der Sprache Fortran geschrieben und ist ein quelloffenes Projekt.

Hingegen ist der `exciting`-Parser in Phyton geschrieben. Sobald mit Hilfe des `exciting`-Codes eine Berechnung durchgeführt wurde, wird unter anderem ein sogenanntes *mainfile* erzeugt. Darin stehen alle wichtigen Informationen, welche auch in das NOMAD-Archiv übernommen werden können. Dazu muss dieses *mainfile* mit Hilfe des `exciting`-Parsers übersetzt werden. Wenn nomad-lab auf dem System installiert ist, auf welchem die Datei übersetzt werden soll, sucht sich nomad-lab selbstständig den richtigen Parser zum Übersetzen der Datei.

3 Stand der Technik

In diesem Kapitel soll der aktuelle Stand der Forschung beleuchtet werden.

3.1 Automatisiertes Metamorphes Testen

Ein Ziel beim Testen von Software ist die Automatisierung. Es soll möglichst wenig Aufwand benötigt werden, um ein Programm zu testen. Daher ist das Automatisieren von Testverfahren ein wichtiges Thema. Es gibt bereits Programme, mit welchen es möglich ist, Software automatisiert metamorph zu testen.

Ein solches Werkzeug ist beispielsweise KABU, welches von Fang-Hsiang Su et. al. in der Arbeit „Dynamic Inference of Likely Metamorphic Properties to Support Differential Testing“ [Fan15] vorgestellt wurde. Mit Hilfe von KABU kann automatisiert nach MR gesucht werden. Dabei soll KABU mehr MR finden als „menschliche“ Entwickler. [vgl. Fan15] Das Programm geht bei der Suche nach MR wie folgt vor:

> First, KABU instruments the system by adding tracing code. As the program executes, KABU generates new inputs and feeds these inputs to sandboxed

> executions of each unit, finally comparing the output states of each of these transformed executions to determine which properties hold. [Fan15]

Im ersten Schritt wird zusätzlicher Programmcode in das zu testende Programm eingefügt, um besser die Transformationen der Eingaben verfolgen zu können. Im zweiten Schritt werden dann weitere Eingaben erzeugt, um MR zu finden, welche mit den ursprünglichen Eingaben nicht gefunden werden können. Daraufhin wird das zu testende Programm mit allen Eingaben (ursprüngliche Eingaben und generierte Eingaben) ausgeführt. Während der Ausführung werden alle Ausgaben gespeichert. Anschließend wird in diesen Daten auf Variablenebene nach MR gesucht. D.h., zu den einzelnen Variablen werden MR gesucht. Für die Suche nach den MR gibt es sechs verschiedene „checker“, welche nach folgenden Relationen zwischen der Ein- und Ausgabe suchen:

- Identitäten ($f(x) = f(x')$),
- Multiplikationen ($f(x) * d = f(x')$),
- Negationen ($f(x) * -1 = f(x')$),
- Additionen ($f(x) + d = f(x')$),
- Änderung der Reihenfolge (wenn die Ein- und Ausgabe ein Array mit denselben Elementen ist, prüfen ob sich die Reihenfolge verändert hat),
- Reverse (ob $f(x)$ das Reverse von $f(x')$ ist)

In der Auswertung des Programms stellte sich heraus, dass KABU auch MR finden konnte, welche durch Studenten nicht gefunden wurden. [vgl. Fan15]

Dieses Werkzeug wäre möglicherweise geeignet gewesen, um die Parser der NOMAD-Codes zu testen. Jedoch ist KABU aufgrund des Alters nicht mehr lauffähig, da einige Bibliotheken, die im Code verwendet wurden, veraltet oder nicht öffentlich zugänglich sind. Jedoch werden wir in unserem Programm einige Ideen von KABU wiederverwenden. Unter anderem das Generieren weiterer Eingaben und die Suche nach MR werden wir ähnlich wie in KABU umsetzen. Jedoch werden wir keinen weiteren Code in das zu testende Programm einbauen.

3.2 Machine-learning Ansätze zum Suchen von MR

Machine-learning ist eine Methode, mit der ohne menschliches Zutun Aufgaben durch den Computer erledigt werden lassen. Machine-learning kann in vielen Bereichen angewendet werden. Auch gibt es verschiedene Methoden, dem Rechner das gewünschte Verhalten „beizubringen“. Beispielsweise Supervised-Learning, hierbei lernt der Algorithmus von Beispielen. Diese sind unter anderem auch mit Labeln versehen. [vgl. Jaf18] Auch zum Suchen von MR kann machine-learning verwendet werden.

Ein solcher Ansatz wird in der Arbeit „Using Machine Learning Techniques to Detect

Metamorphic Relations for Programs without Test Oracles“ [Upu13b] von Upulee Kanewala und James M. Bieman beschrieben. Dabei wird im ersten Schritt aus dem Programm, in welchem die MR gesucht werden sollen, ein Kontroll-Fluss-Graph (CFG) erzeugt. Ein CFG beschreibt den Ablauf eines Programms unabhängig der Programmiersprache als Graph. Aus diesem Graphen können dann Merkmale herausgefiltert werden, mit welchen der machine-lerning Algorithmus ein Modell erzeugt. Dieses Modell kann nun zum Vorhersagen der MR genutzt werden. Als Lernmethode wird „Supervised-learning“ verwendet. Das bedeutet, die verwendeten Trainingsdaten für den Algorithmus sind vollständig mit Labeln versehen. Verglichen werden die beiden machine-learning-Methoden „Decision Trees“ und „Support Vector Machiens“. Es stellte sich heraus, dass sich SVM besser eignen, um MR vorherzusagen. [vgl. Upu13b]
Leider werden in dieser Arbeit keine genaueren Angaben zu den verwendeten Daten gemacht, auch wird kein konkreter Programmcode mitveröffentlicht, was es schwierig macht, diesen Ansatz für unsere Arbeit weiterzuverwenden. Auch wird nicht beschrieben, wie die gefundenen MR aussehen.

In der Arbeit „Using Semi-Supervised Learning for Predicting Metamorphic Relations“ [Bon18] wird von Upulee Kanewala und Bonnie Hardin ein weiter Ansatz vorgestellt, bei welchem ein Semi-Supervised machine-learning Ansatz verwendet wird, um automatisiert nach MR zu suchen. Das heißt, ein Großteil der Daten, welche der Algorithmus zum Lernen verwendet, sind nicht mit Labeln versehen, wie es bei einem Supervised-Ansatz der Fall wäre. Die Daten, auf welchen der ML-Algorithmus angewendet wird, werden wie beim ersten Ansatz auch aus einem CFG des zu testenden Programms extrahiert. Verglichen werden SVM und „Lable Propagation“, wobei sich Letzteres als effizienter herausstellte. [vgl. Bon18]
Das Problem an diesem Ansatz ist es, dass der Algorithmus nur angibt, ob eine der vordefinierten Relationen zutrifft (Addition, Multiplikation, Permutation, Inklusion, Exklusion, Inversion). Was jedoch nicht ausreichend wäre, um auf dem `exciting`-Parser nach Relationen zu suchen.

Ein weiterer ML-Ansatz, bei welchem graph-kernels verwendet werden, um MR zu finden, wird in der Arbeit „Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels“ [Upu15] vorgestellt. An sich basiert auch dieser Ansatz wieder auf dem CFG des zu testenden Programms. Als machine-learning-Algorithus wird eine SVM verwendet. [vgl. Upu15]
Das Problem mit diesem Ansatz ist, die Programme, auf welchen der Algorithmus validiert wurde, sind Funktionen, welche auf numerischen Ein- und Ausgaben basieren. Daher wäre es nicht ohne Weiteres möglich, diesen Ansatz auch für die NOMAD-Parser zu verwenden.

3.3 Testen von wissenschaftlicher Software

Um wissenschaftliche Software strukturiert zu testen, kann man nicht vorgehen wie bei vielen anderen Programmen, denn bei Software, welche im wissenschaftlichen Umfeld

genutzt wird, treten folgende Probleme auf:

Es werden Programme verwendet,

1. welche entwickelt wurden, um Antworten zu finden, welche noch nicht bekannt sind,

2. die zu viele Daten als Ausgabe produzieren, sodass es nicht mehr möglich ist, jede einzelne Ausgabe zu prüfen,

3. bei welchen der Tester eine falsche Vorstellung über das Programm hat.

Diese Probleme wurden in der Arbeit „Techniques for Testing Scientific Programs Without an Oracle“ [Upu13a] von Upulee Kanewala und James M. Bieman aufgelistet. Dennoch gibt es Methoden, solche Programme zu testen. Die folgenden drei Methoden wurden in [Upu13a] vorgestellt und miteinander verglichen: Methamorphes Testen, *run-time assertions*, und durch *machine learning* erzeugte Testorakel.
In der Arbeit von Upulee Kanewala und James M. Bieman, wird ein Experiment zitiert, bei welchem MT gegen „cross-validation“ getestet wird. Das Experiment umfasste zwei Programme, von welchen je 30 Mutanten generiert wurden. Bei beiden Programmen konnten jeweils über 90% der Mutanten durch MT identifiziert werden. Damit stellte sich MT gegenüber „cross validation“ als effektiver heraus. Als Probleme beim MT wurde das Suchen und Auswählen der MR aufgezählt. Auch kann es sein, dass gewisse Fehler selbst durch gute MR nicht entdeckt werden können.
„Run-time assertions“ sind boolesche Ausdrücke, mit welchen unter anderem überprüft werden kann, ob die Ausgabe eines Programms in einem bestimmten Bereich liegt, auch wenn der konkrete Wert nicht bekannt ist oder ob eine bekannte Relation zwischen zwei Variablen eingehalten wurde. Um solche „assertions“ zu finden existieren bereits Werkzeuge, ein Beispiel wäre „Daikon“ [M. 99]. Jedoch wird hier meist noch menschliches Zutun benötigt, damit keine falschen „assertions“ zum Generieren der Testsuite verwendet werden.

4 Implementierung - exciting-parser-testing-tool

In diesem Kapitel werden wir die Entwicklung und den Aufbau des Programms beschreiben, welches wir für diese Machbarkeitsstudie entwickelt haben. In Abbildung 5 ist der grobe Arbeitsablauf zu sehen, wie mit Hilfe der Nomad-Codes Berechnungen durchgeführt und geparst werden. Die zum parsen verwendeten Parser sollen nun metamorph getestet werden. Dazu benötigen wir im ersten Schritt die MR. Diese soll unser Werkzeug automatisiert suchen.

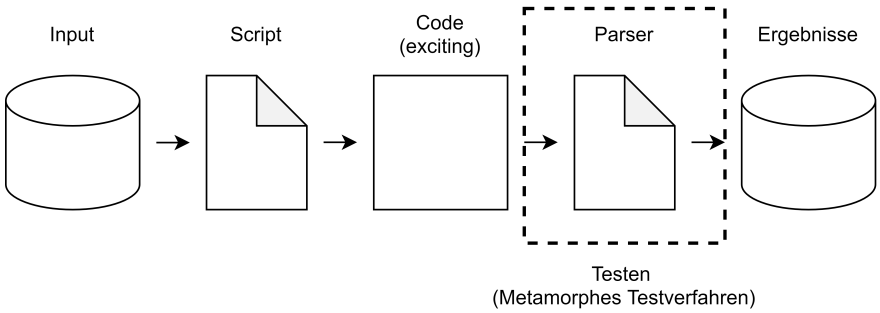

Abbildung 5: Übersicht - NOMAD-Berechnung Arbeitsablauf: Diese Abbildung zeigt den Arbeitsablauf einer Berechnung mit Hilfe eines NOMAD-Codes. Zusehen ist auch der Parser, auf welchem wir in dieser Arbeit nach MR suchen werden.

4.1 Vorwissen

In Abbildung 6 ist eine grobe Übersicht unseres Programms dargestellt. Auf die einzelnen Elemente in dieser Grafik werden wir in diesem Kapitel detailliert eingehen.

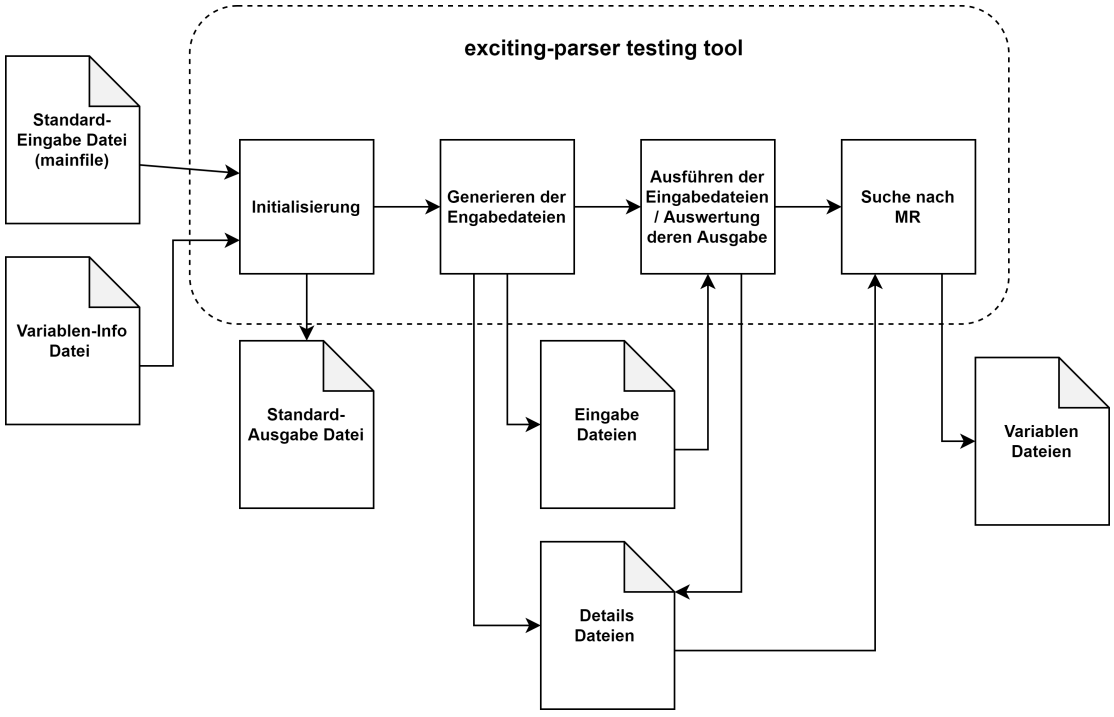

Abbildung 6: Übersicht - exciting-parser-testing-tool - Grob dargestellt ist der Aufbau unseres Programms zur Suche nach MR auf dem `exciting`-Parser. Auch die verwendeten und erzeugten Dateien sind dargestellt.

4.1.1 Verwendete/erzeugte Dateien

Das Werkzeug wird während der Laufzeit mehrere unterschiedliche Dateien anlegen und verwenden. Um einen besseren Überblick über das Werkzeug und dessen Verhalten zu erhalten, werden wir als Erstes auf die verschiedenen Dateien eingehen, welche benötigt und durch unser Werkzeug verarbeitet und erzeugt werden.

Das Programm benötigt die folgenden Dateien, um korrekt funktionieren zu können:

- **Standard-Eingabe Datei** (*mainfile*)
- **Variablen-Info Datei** - Datei mit allen Informationen, zu den in der *Standard-Eingabe Datei* enthaltenen Variablen

Standard-Eingabe Datei: Per Parameter muss der Pfad zu einer *Standard-Eingabe Datei* mitgegeben werden. Jede Berechnung, welche durch einen in NOMAD unterstützten DFT Code durchgeführt wird, besitzt ein sogenanntes *mainfile*. Dieses *mainfile* kann mit Hilfe des zum Code gehörigen Parser übersetzt werden. Ein solches *mainfile* wird von unserem Werkzeug als Grundlage zur Generierung aller weiteren Eingabe Dateien verwendet und muss daher als *Standard-Eingabe Datei* übergeben werden. Während der Entwicklung haben wir das *mainfile*, der unter [3] einsehbaren Berechnung, verwendet. Dort kann auch das *mainfile* heruntergeladen werden.

Allgemein sind die *mainfiles* textbasierte Dateien mit der Endung .out. Ein *mainfile*, welches durch den `exciting`-Code erzeugt wird, ist wie folgt aufgebaut:

1. Kopfzeile
2. Initialisierung
3. Selbstkonsistente Schleife
4. Fußzeile

In der Kopf- und Fußzeile stehen allgemeine Informationen zur Ausführung des Experimentes. Beispielsweise Datum und Uhrzeit, Version des Codes, Zeiten (wie lange die Ausführung gedauert hat) und weitere Informationen. Diese Informationen sind für unser Programm nicht relevant, da diese nicht geparst werden.

Die verschiedenen Blöcke sind optisch durch bestimmte Textblöcke voneinander getrennt. In Abbildung 7 ist ein Beispiel für einen solchen Textblock zu sehen. Dieser Textblock markiert den Beginn des Initialisierung-Abschnittes. Im Block der Initialisierung werden Informationen zur Ausgangssituation aufgelistet. Um einmal ein Beispiel zu nennen, in dem von uns verwendetem *mainfile* ist die, in Abbildung 8 gezeigte Angabe, zu finden.

[3] `https://nomad-lab.eu/prod/rae/gui/entry/id/XTiHVXX2RYmyOmi1Xio-xg/O8EdoGkmRRyjB5B73mYs8iVP1Aas`

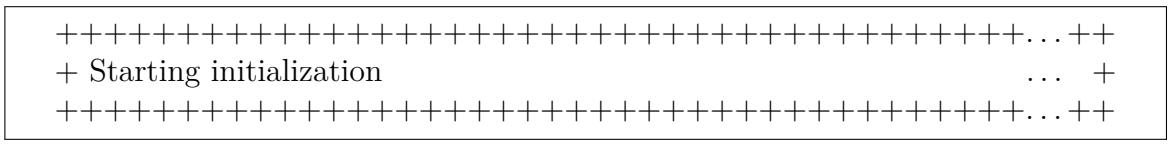

Abbildung 7: Beispiel mainfile - Block

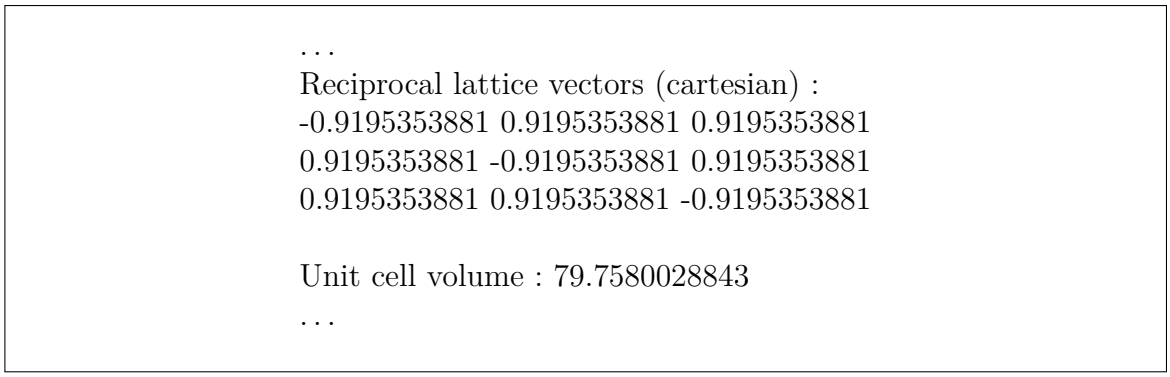

Abbildung 8: Beispiel mainfile

Die Informationen aus dem Block *Initialisierung* sind für unser Werkzeug relevant, denn die Angaben aus diesem Block werden übersetzt und haben somit Einfluss auf die Ausgabe des Parsers.

Im Block der selbstkonsistenten Schleife werden alle Informationen gespeichert, welche während der Berechnung durch den `exciting`-Code erzeugt werden. Es ist möglich, dass es mehrere Schleifendurchläufe gegeben hat. In diesem Fall sind nur die Informationen des letzten Schleifendurchlaufes relevant, da nur diese vom Parser berücksichtigt werden. Die Angabe der Informationen unterscheidet sich nicht von der Art der Angabe, wie bereits im Block der Initialisierung.

Variablen-Info Datei: Um das Werkzeug nutzen zu können, muss noch eine weitere Datei übergeben werden, die *Variablen-Info Datei*. In dieser Datei befinden sich alle Informationen zu den in der *Standard-Eingabe Datei* vorhandenen Variablen. Zu jeder Variable, welche durch das Programm verarbeitet werden soll, müssen folgende Informationen angegeben werden:

- Variablen-Name:
  (z.B. „Fermi energy“)
- Variablen-Typ:
  („int“, „float“, „string“, „vector“ oder ein spezieller Datentyp)
- Block, in welchem die Variable vorkommt:
  („initialization“ oder „last_self_consistent_loop“)

- Falls vorhanden, kann als letzter Parameter eine „Unterliste“ übergeben werden: (z.B. „name_of_sublist“)

Neben den bereits genannten Variablen-Typen gibt es noch Listen, beispielsweise eine Liste von 3 Integer-Werten, wie in Abbildung 9 dargestellt. Diese Liste wäre dann vom Variablen-Typ „list_int_3“. Auf diese Art können ebenso Listen vom Typ Float oder String angegeben werden.

```
G-vector grid sizes      :   24   24   24
```

Abbildung 9: Beispiel Variablen-Type: list_int_3

Angegeben werden diese Informationen zu den Variablen in folgendem Format:

Variable-Declaration: ["Number of crystal symmetries", "int", "initialization"]

Dabei sind folgende Schlüsselworte wichtig:
„Variable-Declaration:“ In der *Variablen-Info Datei* werden beim Einlesen nur Zeilen berücksichtigt, welche mit dem Schlüsselwort „Variable-Declaration:“ beginnen. Darauffolgend werden in eckigen Klammern alle weiteren Informationen zu der Variable angegeben.
„sublist: Name der Unterliste“ Wenn in den eckigen Klammern als erstes Argument eine Unterliste angegeben wird, dann wird die Variable der genannten Unterliste hinzugefügt. Diese Unterliste kann einer anderen Variable hinzugefügt werden.
Mithilfe des „#“ kann eine Zeile mit Informationen zu Variablen auskommentiert werden, diese wird dann beim Einlesen nicht berücksichtigt.

Eine Variablen-Unterliste sollte dann erstellt werden, wenn mehrere Variablen eindeutig zu einem Variablen-Block zugeordnet werden können. In Abbildung 10 ist ein Beispiel für einen solchen Variablen-Block zu sehen. In der darauf folgenden Abbildung 11 ist eine Möglichkeit dargestellt, die Variablen-Information für das Werkzeug bereitzustellen.

```
Maximum angular momentum used for
        APW functions                    :    8
        computing H and O matrix elements :   8
        potential and density            :    8
        inner part of muffin-tin         :    2
```

Abbildung 10: Standard-Eingabe Datei

Bis jetzt muss diese Datei händisch erzeugt werden. Ein Thema für eine spätere Arbeit wäre es, die Informationen zu den Variablen direkt aus dem *mainfile* zu lesen.

Zur Erstellung der *Variablen-Info Datei* sind wir, wie in Abbildung 12 zu sehen, vorgegangen. Die Zeilen aus dem ursprünglichen *mainfile* sind weiterhin in der *Variablen-Info*

```
Variable-Declaration: ["sublist: Maximum angular momentum",
                       "APW functions", "int", "initialization"]
Variable-Declaration: ["sublist: Maximum angular momentum",
                       "computing H and O matrix elements", "int", "initialization"]
Variable-Declaration: ["sublist: Maximum angular momentum"
                       , "potential and density", "int", "initialization"]
Variable-Declaration: ["sublist: Maximum angular momentum",
                       "inner part of muffin-tin", "int", "initialization"]
Variable-Declaration: ["Maximum angular momentum used for",
                       "none", "initialization", "Maximum angular momentum",]
```

Abbildung 11: Variablen-Info Datei

Datei vorhanden. Dies ist nicht nötig, da beim Einlesen der Variablen nur Zeilen berücksichtigt werden, welche mit dem Schlüsselwort „Variable-Deklaration:“ beginnen, jedoch macht es die *Variablen-Info Datei* übersichtlicher.

```
Number of Bravais lattice symmetries      :       48
Number of crystal symmetries              :       24

k-point grid                              :       8    8    8
```

mainfile

```
Number of Bravais lattice symmetries      :       48
Variable-Declaration: ["Number of Bravais lattice symmetries", "int", "initialization"]
Number of crystal symmetries              :       24
Variable-Declaration: ["Number of crystal symmetries", "int", "initialization"]

k-point grid                              :       8    8    8
Variable-Declaration: ["k-point grid", "list_int_3", "initialization"]
```

mainfile mit Informationen zu den Variablen

Abbildung 12: Erstellen der Variablen-Info Datei

Eingabe Datei: Als *Eingabe Datei* bezeichnen wir alle Dateien („Input-Files“), welche auf Basis der *Standard-Eingabe Datei* durch unser Werkzeug generiert werden. Die *Eingabe Datei* und die *Standard-Eingabe Datei* sind bis auf eine Variable (die Variable, für welche diese Datei generiert wurde) identisch.

Ausgabe Datei (Stdout- und Stderr-Ausgabe Datei): Als *Ausgabe Datei* bezeichnen wir alle Dateien, welche durch die Ausführung des Parsers entstehen. Normalerweise

schreibt der `excitng`-Parser das übersetzte *mainfile* auf die *Standard-Ausgabe* des Kommando-Fensters. Unser Programm speichert einerseits die *Standard-Ausgabe* und die Fehler-Ausgabe des Parser in zwei separate Dateien.

Im Laufe dieser Arbeit werden wir die beiden Ausgabe-Dateien, welche beim Parsen der *Standard-Eingabe* entstehen, als *Standard-Ausgabe Dateien* bzw. nur als *Standard-Ausgabe* bezeichnen.

Details Datei: Informationen, welche für die Suche nach MR nützlich sein könnten, werden in einer Details-Datei abgespeichert. Zu jeder Eingabe-Datei wird auch eine Details-Datei generiert und befüllt. Informationen, welche in dieser Datei gespeichert werden, sind unter anderem Informationen, über die in der Eingabe-Datei veränderten Variablen-Werte. Auch die Unterschiede zwischen der *Standard-Ausgabe* und den generierten Ausgabe-Dateien werden in der Details-Datei abgespeichert.

Variablen Datei: Zu jeder Variable existieren mehrere Eingabe- und dementsprechend auch mehrere Ausgabe-Dateien. In der Variablen-Datei werden alle Informationen der einzelnen Ausgabe bzw. *Details Dateien* zusammengefasst, welche einer Variable zugeordnet werden können. Wenn MR gefunden werden, werden diese auch in der Variablen-Datei abgespeichert.

Eine solche *Variablen Datei* sieht wie in Abbildung 13 dargestellt aus.

Log Datei: Die Log-Ausgabe, welche unser Werkzeug generiert, wird automatisch als Log-Datei abgespeichert, um nachträglich den Programmablauf nachvollziehen zu können.

4.1.2 Parameter

Unser Programm lässt sich mithilfe von Python und dem in Abbildung 14 dargestelltem Befehl starten. In Abbildung 14 ist zu sehen, dass einige Parameter übergeben werden müssen und noch weitere Parameter optional übergeben werden können. Weitere Informationen zu den einzelnen Parametern und deren Funktionen sind im Git-Repository[4] zu dieser Arbeit zu finden.

4.1.3 Verwendete Software und Software-Pakete

Bevor unser Programm verwendet werden kann, muss sichergestellt werden, dass folgende Programme und Software-Pakete auf dem verwendetem System installiert sind. Auch die richtige Version ist wichtig, da wir unser Werkzeug nur mit den angegebenen

[4] `https://gitlab.informatik.hu-berlin.de/se/ba-gogoll-valentin`

[‘Unit cell volume‘, ‘80.7580028843‘,
"[‘section_run‘][0][‘section_system‘][0][‘x_exciting_unit_cell_volume‘]",
‘1.1967101356544402e-29‘]
[‘Unit cell volume‘, ‘78.7580028843‘,
"[‘section_run‘][0][‘section_system‘][0][‘x_exciting_unit_cell_volume‘]",
‘1.1670731933598435e-29‘]
[‘Unit cell volume‘, ‘2.0‘,
"[‘section_run‘][0][‘section_system‘][0][‘x_exciting_unit_cell_volume‘]",
‘2.9636942294596794e-31‘]
...

Path:
[‘section_run‘][0][‘section_system‘][0][‘x_exciting_unit_cell_volume‘]
[-1.2089258196146292e+24, ‘-1.791443237718345e-07‘]
[-1.152921504606847e+18, ‘-1.7084534051116418e-13‘]
[-1099511627776.3333, ‘-1.6293081332322688e-19‘]
...

Path:
[‘section_metadata‘][‘dft‘][‘crystal_system‘]
[82.09133621763333, ‘Error‘]

Wrong Values
[‘one‘, ‘Error‘]
[‘‘, ‘Error‘]
[‘"1"‘, ‘Error‘]

Abbildung 13: Beispiel: Variablen Datei

usage: main.py [-h]
-d PATH_TO_STANDARD_INPUT_FILE
-v PATH_TO_STANDARD_INPUT_FILE_WITH_VARIABLES_DECLARATION
-f PATH_TO_FILE_DIRECTORY
[-c COMMAND_PARSER]
[–dry_run]
[-a–accuracy ACCURACY]
[–clean]

Abbildung 14: Aufruf - exciting-parser-testing-tool - zu sehen sind alle Parameter, welche unser Werkzeug verarbeiten kann.

Tabelle 1: Tabelle: Verwendete Programme und Software-Pakete. Dabei steht Version(L) für die verwendete Version unter Linux und Version(W) steht dementsprechend für die unter Windows verwendete Version.

Name	Version(L)	Version(W)	Name	Version(L)	Version(W)
python	3.6.15	3.9.6	deepdiff	5.7.0	5.7.0
nomad-lab	0.10.4	-	argparse	1.1	1.1
ast	82160	-	z3	-	0.2.0
json	2.0.9	2.0.9	z3-solver	4.8.17	4.8.12.0
logging	0.5.1.2	0.5.1.2	platform	1.0.7	1.0.8

Versionen getestet haben. In Tabelle 1 sind alle verwendeten Programme und Pakete aufgelistet.

4.2 Aufbau des Programms

Das Werkzeug lässt sich grob in drei große Phasen aufspalten:

Abschnitt 1: Der erste Part des Programms beschäftigt sich mit dem Generieren, Ausführen und Auswerten der Testdateien, welche durch den Parser übersetzt werden.

Abschnitt 2: Nachdem alle *Eingabe Dateien* ausgewertet wurden, werden alle Informationen zu den einzelnen Variablen zusammengetragen und in den *Variablen Dateien* abgespeichert.

Abschnitt 3: Auf den im vorherigen Schritt zusammengetragenen Daten soll das Werkzeug nach MR suchen. Da das Programm nicht besonders viele Werte für die einzelnen Variablen erzeugen kann – anderenfalls würde die Laufzeit durch das Parsen der einzelnen Dateien extrem ansteigen – ist die Suche nach den MR sehr sensibel, sodass möglichst viele Anhaltspunkte für MR gefunden werden.

4.2.1 Generieren, Ausführen und Auswerten der Testdaten

Initialisierung Im ersten Schritt werden alle übergebenen Parameter verarbeitet. Dazu nutzen wir das Paket argparse[5]. Die übergebenen Werte werden in der „constans-Klasse“ für die Dauer der Laufzeit gespeichert. Ebenso wird überprüft, ob das Programm auf einem Linux oder Windows-System läuft, diese Info ist wichtig für die Angabe der Pfade. Somit ist unser Werkzeug auf Windows und auf Linux lauffähig. Getestet haben wir das Programm unter Windows 10 sowie openSUSE Leap 15.3 in der Kernel-Version 5.3.18 (5.3.18-150300.59.68-preempt).

Um den Programmablauf nachvollziehen zu können haben wir einen Logger verwendet.

[5] `https://docs.python.org/3/library/argparse.html`

Wir verwenden das Paket logging[6]. Dieser Logger wird im ersten Abschnitt unseres Programms initialisiert und konfiguriert.

Nun wird die benötigte Ordnerstruktur angelegt, in welcher unser Werkzeug alle verwendeten und generierten Dateien ablegen kann. Dazu wird der Pfad verwendet, welcher per Parameter übergeben wurde. Wenn der übergebene Pfad auf ein leeres Verzeichnis verweist, werden die folgenden 4 Ordner angelegt:

- input - hier werden alle *Eingabe-Dateien* abgelegt,
- details - alle *Details Dateien* (4.1.1) und *Variablen Dateien* (4.1.1) werden in diesem Ordner abgelegt,
- output - in diesem Ordner werden alle *Ausgabe Dateien* (4.1.1) gespeichert,
- log-files - hier werden alle Log Dateien (4.1.1) gespeichert.

Die Ordnerstruktur ist in Abbildung 15 dargestellt. Sollten diese Ordner existieren und bereits Dateien vorhanden sein, werden diese gelöscht, wenn der Parameter „–clean“ übergeben wurde. Ist dies nicht der Fall, bleiben alle vorhandenen Dateien erhalten.

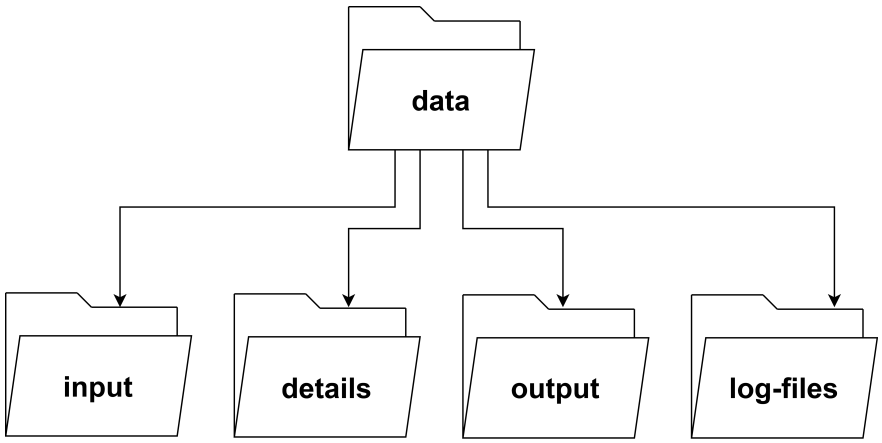

Abbildung 15: Ordnerstruktur - Diese Abbildung zeigt die Ordnerstruktur, welche unser Programm anlegt. Nachdem unser Programm erfolgreich gelaufen ist, finden sich alle wichtigen Dateien, in welchen die gefunden MR aufgelistet werden, im Ordner *details*.

Auslesen der Informationen zu den Variablen Sobald während der Initialisierung der Pfad zu der *Variablen-Info Datei* übergeben wird, wird diese ausgelesen und die darin vorhandenen Informationen zu den Variablen abgespeichert. Den Aufbau der Varibalen-Info Datei haben wir bereits in Abschnitt 4.1.1 ausführlich beschrieben. Neben den in dieser Datei übergebenen Informationen beinhaltet unser Objekt (objektorientierte

[6] `https://docs.python.org/3/howto/logging.html`

Programmierung) für die Variablen noch weitere „Eigenschaften“. Zusätzlich wird ein Variablen-Name generiert, in welchem alle Sonderzeichen durch „_“ ersetzt werden, um diesen Namen für die Dateinamen zu verwenden, welche im Zusammenhang mit jener Variable stehen. Alle einzelnen Variablen-Objekte werden in einer Liste gespeichert.

Ausführung der Standard-Eingabe Datei Die *Standard-Eingabe Datei* (4.1.1) dient als Grundlage zum Generieren der *Eingabe Dateien* (4.1.1) und auch, um die Ausgabe, welche der Parser aus dieser Datei erzeugt, mit den Ausgaben der generierten *Eingabe Dateien* zu vergleichen.

Im ersten Schritt wird die übergebene *Standard-Eingabe* in das Verzeichnis des Programms kopiert und dort mit Hilfe des Parsers übersetzt. Die resultierende Ausgabe, welche in Textform in der Konsole zurückgegeben wird, wird abgespeichert. Ebenso die StdErr-Ausgabe, welche jedoch leer sein sollte.

Die Nomad-Parser können mit folgendem Befehl verwendet werden:

```
nomad parse –show-archive <path-to-mainfile>
```

Abbildung 16: nomad parse Befehl - Dieser Befehl übersetzt das angegebene *mainfile* und schreibt das Ergebnis in die Standard-Ausgabe (wenn nomad-lab installiert ist).

Wenn nomad-lab installiert ist, kann dieser Befehl einfach über die Konsole ausgeführt werden. Dabei sucht sich nomad-lab eigenständig den richtigen Parser. Durch „-show-archive“ wird die übersetzte Datei in der Konsole ausgegeben.

Unser Werkzeug nutzt die Python-Funktion „os.system(command)“, um den Parser zu verwenden. Zusätzlich zu dem Befehl in Abbildung 16 benötigen wir noch die Umleitung der Ausgabe, was zu folgendem Befehl führt:

```
nomad parse –show-archive "path_to_input-file "> "path_to_stdout-file "2>
"path_to_stderr-file "
```

Abbildung 17: Befehl zum Übersetzen der Eingabe Dateien - Der Unterschied zu Abbildung 16 ist, dass bei diesem Befehl die StdOut- sowie StdErr-Ausgabe in separate Dateien abgespeichert werden.

Damit wird die StdOut- und StdErr-Ausgabe in die beiden angegebenen Dateien umgeleitet.

Sollte der „nomad parse“-Befehl bei einem System abweichen, kann per Parameter ein anderer Befehl zum Ausführen des Parsers übergeben werden.

Generieren der Eingabe Dateien Nachdem die *Standard-Eingabe Datei* durch den Parser übersetzt wurde, beginnt das Programm damit, die *Eingabe Dateien* zu generieren. Dazu wird über die Liste aller Variablen iteriert. Zu jeder Variable werden, je nach Typ, mehrere *Eingabe Dateien* generiert. Für den Typ „int“ werden, beispielsweise zu den in Abbildung 18 dargestellten Szenarien, jeweils mehrere *Eingabe Dateien* generiert.

a) Wert der Variable vergrößert sich

b) Wert der Variable verringert sich

c) Wert der Variable geht gegen ∞

d) Wert der Variable geht gegen $-\infty$

e) es werden die Werte 0 und der negierte Ursprungswert eingesetzt

f) es werden fehlerhafte Werte eingesetzt - beispielsweise eine Zeichenkette

Abbildung 18: Wertgenerierung - Szenarien - Diese Szenarien kommen bei der Generierung der Werte für den Datentyp Integer und Float zur Anwendung. Einzelne Szenarien werden auch bei anderen Datentypen verwendet.

Die *Eingabe Dateien* werden generiert, indem die *Standard-Eingabe Datei* kopiert wird, abgesehen von der Zeile, in welcher sich die Variable befindet, welcher gerade bearbeitet wird. In dieser Zeile wird der Wert für die Variable verändert.
Die Genauigkeit, welche per Parameter gesetzt werden kann, entscheidet darüber, wie viele *Eingabe Dateien* zu den einzelnen Szenarien generiert werden. Wenn kein spezieller Wert übergeben wird, werden standardmäßig 5 Dateien zu den einzelnen Szenarien generiert, abgesehen von Punkt e und f aus Abbildung 18.
Zu jeder *Eingabe Datei* wird auch eine *Details Datei* erzeugt. In dieser Datei werden unter anderem die Änderungen an der *Standard-Datei* gespeichert.

In den folgenden Paragraphen werden wir genauer die Generierung der einzelnen Werte für die verschiedenen Variablen-Typen beschreiben:

Wertgenerierung - Integer und Float Für die Variablen-Typen Integer und Float werden *Eingabe Dateien*, für die bereits in Abbildung 18 aufgezählten Szenarien, generiert.

Das Szenario, dass „der Wert der Variable vergrößert sich“, wird durch einfaches Addieren umgesetzt. Dabei vergrößert sich der Wert der Variable immer um 1. Beispielsweise, wenn der ursprüngliche Wert der Variable 5 ist und drei weitere *Eingabe Dateien* erzeugt werden, dann werden in den generierten *Eingabe Dateien* die Werte 6, 7, und 8 eingesetzt.

Die Generierung der Float-Werte findet auf demselben Weg statt, nur dass bei den Float-Werten zusätzlich ein Wert kleiner als 1 dazu addiert wird, damit sich auch der Wert hinter dem Komma verändert. Konkret sieht die Rechnung wie folgt aus:

Integer-Werte:	Neuer Wert = Ursprünglicher Wert + i
Float-Werte:	Neuer Wert = Ursprünglicher Wert + $i + 1/(i+1)$

Dabei steht i für die Nummer der Eingabe Datei
welche zu diesem Szenario generiert wird.

Abbildung 19: Wertgenerierung - Integer und Float: Dargestellt ist die Formel, mit welcher die Werte für das Szenario der steigenden Werte berechnet werden.

Um die Werte für das Szenario zu generieren, in welchem sich die Werte der Variable verringern, gehen wir genauso vor, bis auf dass folgende Formeln verwendet werden:

Integer-Werte:	Neuer Wert = Ursprünglicher Wert − i
Float-Werte:	Neuer Wert = Ursprünglicher Wert − $i - 1/(i+1)$

Abbildung 20: Wertgenerierung - Integer und Float: Dargestellt ist die Formel, mit welcher die Werte für das Szenario der fallenden Werte berechnet werden.

Für die Berechnung der Werte gegen ∞ (bzw. -∞) haben wir eine exponentiell steigende Funktion verwendet, siehe Abbildung 21.

Alle Werte, welche wir generieren, liegen in der Funktion 2^i. Um in gleichen Abständen Werte aus dieser Funktion zu entnehmen, haben wir die Variable „step_size“ definiert und den maximalen Wert für i mit 100 definiert. Die Idee ist, wenn wir beispielsweise 3 Dateien generieren wollen, ist der erste Wert $2^{(1*(100/3))}$ also 2^{33}, der zweite Wert wäre demzufolge 2^{66} und der dritte 2^{100}. Somit können wir einen relativ großen Wertebereich abdecken.

Als spezielle Werte verwenden wir für Integer und Float folgende Werte: 0, 1, -(Ursprünglicher Wert). Diese Werte bilden nur eine kleine Grundlage. Die Idee ist, diese Werte durch eine Parameter-Eingabe noch zu erweitern, beispielsweise, wenn bereits Werte für bestimmte Variablen bzw. Werte bekannt sind, bei welchen sich der Parser besonders verhalten soll. Diese Funktionalität ist jedoch noch nicht implementiert. In Abschnitt 5.5.1 gehen wir noch einmal genauer auf diese Idee ein.

Um *Eingabe Dateien* zu generieren, welche möglicherweise zu einem Fehler führen könnten, verwenden wir vordefinierte fehlerhafte Werte, siehe Abbildung 22. Unter anderem verwenden wir anstelle eines Zahlenwerts eine Zeichenkette oder eine leere Eingabe. Sollten bestimmte Eingaben tatsächlich zu einem Fehler führen, lassen sich auch daraus MR ableiten.

$$
\begin{aligned}
i &= \text{aktuelle Nummer der Datei} \\
accuracy &= \text{maximale Anzahle zu generierenden Dateien} \\
stepSize &= 100/accuracy \\
\\
\text{Wert} &= 2^{(i*stepSize)}
\end{aligned}
$$

Abbildung 21: Wertgenerierung - Integer und Float - zu sehen ist die Formel, mit welcher die Werte für die Szenarien c und d aus Abbildung 18 generiert werden.

```
input_values_for_int_invalid = ["one", 1.212, "inf", "", "1", "1.123"]
input_values_for_float_invalid = ["one", 100, "inf", "", "1", "1.123"]
```

Abbildung 22: Wertgenerierung - Integer und Float - Dargestellt sind die vordefinierten, fehlerhaften Werte, welche wir in unserem Programm verwenden.

Wertgenerierung - String Um neue Eingabewerte vom Typ String zu generieren, verwenden wir drei verschiedene Methoden:

Verwenden vordefinierter Eingabewerte (korrekte Werte): Hier verwenden wir vordefinierte Werte. Diese Werte sind bis jetzt folgende: "t", "test"und "test test". Die Idee hinter diesen Eingabewerten ist, herauszufinden ob es ausreicht einen einzelnen Buchstaben anzugeben, wie der Parser ein Wort verarbeitet und ob der Parser auch mit Leerzeichen in einem String umgehen kann. Es wäre eine gute Idee, eine Möglichkeit zu schaffen, diese Eingabewerte noch erweitern zu können, denn unsere Werte decken nur einen sehr kleinen Bereich von interessanten Eingabewerten ab.

Verwenden vordefinierter Eingabewerte (fehlerhafte Werte): Auch bei dieser Methode verwenden wir wieder vordefinierte Werte. Diesmal jedoch Werte, welche nicht unbedingt der Definition eines Strings entsprechen. Daher verwenden wir folgende Werte: 1, „“, und „“““. Die Idee hinter diesen Werten ist, herauszufinden, wie sich der Parser verhält, sobald anstatt eines Strings ein Integer-Wert übergeben wird und wie der Parser mit der leeren Eingabe umgeht. Dabei verwenden wir einerseits die beiden Anführungszeichen als leere Eingabe, aber auch eine vollkommen leere Eingabe.

Generieren zufälliger Strings: Als letzte Methode generieren wir uns, je nachdem wie hoch die Genauigkeit für unser Programm festgelegt wurde, mehrere zufällige Eingaben. Das Wichtige dabei ist, der erste Eingabewert ist zwei Zeichen lang, der 3 Eingabewert ist dann dementsprechend 4 Zeichen lang. Der Grund hierfür ist, heraus-

zufinden, ob der Parser je nach Länge der Eingaben sich unterschiedlich verhält. Um zufällige Buchstaben auszuwählen, verwenden wir die von Python, bzw der Bibliothek „random“[7], bereitgestellte Funktion: *random.choice(letters)*.

Um vergleichbare Ergebnisse zu generieren, verwenden wir für die Auswertung anstelle der *random.choice(letters)* Funktion, vordefinierte Strings. Diese Strings haben wir dennoch zufällig generiert.

Wertgenerierung - Vektor Für den Variablen-Typen Vektor verwenden wir drei verschiedene Szenarien. Die ersten beiden Szenarien entsprechen den Szenarien a und b aus Abbildung 18. Nur werden bei den Vektoren alle Werte mit den Formeln in Abbildung 19 und Abbildung 20 berechnet.

Das letzte Szenario bei der Generierung der Werte für Vektoren entspricht dem Szenario e aus Abbildung 18. Hier wird auch wieder jeder Wert im Vektor einzeln betrachtet.

Allgemein zu der Wertgenerierung für Vektoren ist zu sagen, dass wir nicht mehr zwischen Integer- und Float-Werten unterscheiden. Alle Werte werden als Float betrachtet.

Wertgenerierung – Listen (vom Type Integer oder Float) Um weitere Eingabewerte für Listen vom Type Integer oder Float zu generieren verwenden wir vier verschiedene Szenarien.

Steigende und fallende Werte: Wie auch bei den Werten für Integer und Float vergrößern bzw. verkleinern wir die Eingabewerte. Dazu gehen wir durch die Liste und berechnen jeden einzelnen Wert genauso, wie in Abbildung 19 und Abbildung 20 bereits beschrieben.

Spezielle Werte: Auch bei den Listen verwenden wir bereist vordefinierte, spezielle Werte. Im ersten Fall ersetzen wir alle Werte der Liste mit 0. Für den zweiten Fall negieren wir alle Werte der Liste.

Fehlerhafte Werte: Um das Verhalten des Parsers zu untersuchen, wenn keine korrekten Werte übergeben werden, setzen wir anstelle der Liste einen einzelnen Wert ein. Um zu prüfen, ob der Parser mit einem einzelnen, aber dennoch korrekten Wert (vom Typ Integer oder Float) umgehen kann, setzen wir an dieser Stelle entweder 123 als Integer oder 123.123 als Float ein. Um zu prüfen, was bei einer vollkommen falschen Eingabe passiert, setzen wir im nächsten Schritt den String „test“ als Eingabe ein.

Veränderungen der Länge der Liste: Als letztes Szenario betrachten wir den Fall, dass sich die Länge der Liste verändert. Dazu verwenden wir die Liste aus der *Standard-Eingabe* und kürzen bzw. erweitern die Liste. Um die Liste zu kürzen, schneiden wir die letzten Elemente an. Um die Liste zu verlängern, fügen wir Werte (als Wert verwenden wir das erste Element der Liste) ans Ende der Liste an.

[7] `https://docs.python.org/3/library/random.html`

Ausführen der Eingabedateien Die Ausführung der Eingabedateien findet in einer Schleife statt, sodass alle im vorherigen Schritt generierten *Eingabe Dateien* der Reihe nach geparst werden. Dabei wird derselbe Befehl verwendet, wie bereits in Abbildung 17 vorgestellt.

Auswerten der übersetzten Dateien Nachdem alle *Eingabe Dateien* ausgeführt wurden, werden diese gegen die Ausgabe der *Standard-Eingabe Datei* verglichen. Der Vergleich findet auf JSON-Ebene statt, da sich die Parameterreihenfolge in der Ausgabedatei häufig unterscheidet. Dies stellt bei einem JSON-Vergleich kein Problem dar. Für den Vergleich auf JSON-Basis nutzen wir das Paket DeepDiff[8]. Alle gefundenen Unterschiede werden in der *Details Datei* gespeichert. Im Idealfall gibt es nur wenige Unterschiede zwischen den *Ausgabe Dateien*. Dies macht die Suche nach MR einfacher. Auch wird geprüft, ob die StdErr-Ausgabe leer ist oder ob es zu einem Fehler gekommen ist. Wurde ein Unterschied zwischen den Dateien festgestellt, wird neben dem alten und dem neuen Wert auch der JSON-Pfad gespeichert, an welcher Stelle der Unterschied aufgetreten ist. Ähnlich wie ein Datei-Pfad gibt ein JSON-Pfad an, an welcher Stelle in einer JSON-Datei eine bestimmte Information steht. Ein Beispiel für eine *Details Datei* ist in Abbildung 23 zu sehen. Bei dieser *Eingabe Datei* ist in der Ausgabe nur ein Unterschied, im Vergleich zur *Standard-Ausgabe*, aufgetreten.

```
Differences in Input-Files:
Standard-File: Unit cell volume : 79.7580028843
Current-File: Unit cell volume : 83.9580028843

compare stdout-file

path:
['section_run'][0]['section_system'][0]['x_exciting_unit_cell_volume']
old: key: x_exciting_unit_cell_volume value: 1.1818916645071418e-29
new: key: x_exciting_unit_cell_volume value: 1.2441292433257951e-29

compare stderr-file
Keinen Fehler gefunden
```

Abbildung 23: Beispiel: Details Datei

Der Grund, weshalb pro Datei nur eine Änderung der *Standard-Eingabe Datei* vorgenommen wird ist, dass so die Änderungen, welche sich in den *Ausgabe Dateien* finden lassen, zu genau einer Änderung der *Eingabe Datei* zugeordnet werden können. So ist es möglich, auf diesen Daten nach MR zu suchen. Es ist jedoch nicht möglich nach MR zu suchen, bei denen zwei oder mehr Eingabe-Variablen verändert werden. Dies könnte

[8] `https://deepdiff.readthedocs.io/en/latest`

als Thema in weiteren Arbeiten behandelt werden.

Im nächsten Schritt werden alle Informationen zu den ausgeführten Dateien zusammengefasst, welche im Zusammenhang mit einer bestimmten Variable stehen. Für jede Variable, welche durch das Werkzeug untersucht wird, wird eine weitere Datei erstellt (*Variablen Datei*, siehe Abschnitt 4.1.1). In dieser Datei werden die Ein- und Ausgabewerte der einzelnen ausgeführten *Eingabe Dateien* zusammengetragen. Dies erleichtert es, in den weiteren Schritten nach MR zu suchen.

Im ersten Abschnitt der Datei werden alle Unterschiede zusammengefasst, welche bei den ausgeführten *Eingabe Dateien* aufgetreten sind, die im Zusammenhang mit der Variable stehen, zu welcher die *Variablen Datei* gehört. Daraufhin werden alle Unterschiede aufgelistet, welche in den einzelnen Pfaden der Ausgabe aufgetreten sind. Letztendlich werden alle Eingaben aufgelistet, bei welchen es zu einem Fehler gekommen ist.

4.2.2 Suchen nach metamorphen Relationen

Die Suche nach den MR findet auf den zuvor generierten *Variablen-Info Dateien* statt. Hier gibt es nun für jeden Daten-Typen wieder einzelne Methoden um MR zu finden.

Integer und Float: Die Suche nach MR bei Variablen vom Typ Integer und Float läuft ähnlich ab, wie bei KABU (Unterabschnitt 3.1). In unserem Werkzeug existieren dazu 6 Funktionen. Jede dieser Funktionen ist in der Lage, eine bestimmte Art von MR zu finden. Auf die einzelnen Funktionen gehen wir im Folgenden ein:

Identitäten: Die erste Funktion sucht nach Ein-Ausgabe Paaren, bei welchen die Eingabe gleich der Ausgabe ist. Wenn beispielsweise in der *Eingabe Datei* die Variable „Unit cell volumen“ den Wert 80.5 trägt und auch in der *Ausgabe Datei* die Variable denselben Wert erhält, dann würde eine Identität (identische Abbildung) vorliegen.

Je nachdem, wie viele Identitäten unser Programm finden kann, lässt sich daraus eine MR ableiten. Wir unterscheiden, ob für alle, fast alle oder nur für einzelne Eingabewerte eine Identität vorliegt. Das heißt, wir haben die MR „Es liegt eine Identität vor“ in drei spezifischere MR aufgeteilt.

Wenn die Eingabewerte steigen, steigen auch die Ausgabewerte: Diese Funktion überprüft, ob die Ausgabewerte größer werden, wenn auch die Eingabewerte größer werden. Wir betrachten dabei jedoch nicht alle Eingabewerte mit einem Mal, sondern durchlaufen die sortierte Liste der Ausgabewerte. Sobald 3 aufeinanderfolgende Ausgabewerte gestiegen sind, speichern wir diesen Fund als MR ab. Somit können wir auch MR finden, wenn ein bestimmtes Monotonieverhalten nur in einem bestimmten Wertebereich auftritt. Auch unterscheidet diese Funktion, ob die Werte monoton oder streng monoton steigen.

Wenn die Eingabewerte steigen, fallen die Ausgabewerte: Diese Funktion untersucht, genau wie die letzte Funktion, das Monotonieverhalten der Ausgabewerte.

Daher unterscheiden sich beide Funktionen kaum, bis auf dass nach fallenden und nicht nach steigenden Werten gesucht wird.

Verhalten der Ausgabewerte, wenn die Eingabewerte negiert werden: Diese Funktion prüft, ob die Aussage in Abbildung 24 erfüllt wird. Auch hier wird wieder unterschieden, ob die Aussage für alle, fast alle oder nur spezielle Eingabewerte zutrifft.

Einagbewert => Ausgabewert, dann auch: -Einagbewert => -Ausgabewert

Abbildung 24: Negierung - Wenn diese Aussage erfüllt ist, könnte eine MR vorliegen.

Addition/Multiplikation konstanter Werte: Die letzte Funktion prüft, ob konstante Werte zu den Eingabewerten addiert oder multipliziert wurden. Für diese Prüfung verwendeten wir den Z3[9] „Theorembeweiser“ (eng. theorem prover). Wenn dieser Theorembeweiser nun feststellen kann, dass zu allen Eingabewerten ein bestimmter Wert hinzuaddiert bzw. jeder Eingabewert mit einem bestimmten Wert multipliziert wurde, lässt sich daraus auch eine MR ableiten.

Vektoren: Für die Suche nach MR bei Variablen vom Type Vektor, ist bisher nur eine Such-Funktion in unserem Programm implementiert. Bisher lassen sich nur Identitäten suchen, das heißt, nur bei Eingaben, bei welchen in der Ausgabedatei der Eingabe-Vektor zu finden ist, wird eine MR gefunden.

Identitäten: Die Suche nach Identitäten haben wir so umgesetzt, wenn in der Ausgabe-Datei eine Änderung gefunden wurde und diese sich als Vektor identifizieren lässt, vergleichen wir jeden Wert dieser Vektoren (umgewandelt als Float) miteinander. Damit wollen wir ausschließen, dass durch unterschiedliche Formatierungen oder fehlende Nachkommastellen (gemeint ist hier beispielsweise 1.0 verglichen mit 1) MR nicht gefunden werden.

Jedoch sollten zu den Vektoren weitere Methoden umgesetzt werden, um auch andere MR finden zu können.

Strings: Um auf Strings nach MR zu suchen, verwenden wir drei verschiedene Funktionen.

Identität: Wie bereits bei anderen Variablen-Typen, suchen wir auch auf Strings wieder nach Identitäten. Dazu vergleichen wir einfach die Ein- und Ausgabe, was problemlos funktioniert.

Parser-verhalten bei speziellen Eingaben: Diese Kategorie bezieht sich auf das Verhalten der Parser, wenn beispielsweise Leerzeichen in der Eingabe enthalten sind oder anstatt eines Strings ein Integer-Wert übergeben wird. Genau diese beiden Fälle

[9] `https://github.com/Z3Prover/z3`

prüft unser Werkzeug und je nachdem, ob der Parser Eingaben mit Leerzeichen oder Zahlenwerte verarbeiten kann, lässt sich daraus eine MR ableiten. Jedoch sollten noch weitere Methoden implementiert werden, um besser nach MR zu suchen.

Listen: Unser Programm verwendet derzeit zwei Methoden, um auf Listen nach MR zu suchen.

Identität: Wie bereist bei allen anderen Datentypen suchen wir auch auf Listen nach Identitäten. Dazu vergleichen wir die einzelnen Werte der Listen miteinander. Der Grund dafür ist, so können wir die Werte als Integer bzw. Float verarbeiten und verhindern, dass Identitäten aufgrund unterschiedlicher Darstellung oder Formatierung nicht gefunden werden.

Länge der Listen: Ebenso prüfen wir, ob sich die Länge der Ein- und Ausgabelisten unterscheiden.

Alle gefundenen MR werden in die verschiedenen Variablen-Dateien geschrieben. Nachdem auf allen Variablen nach MR gesucht wurde, endet unser Programm. Bisher fehlt noch die Validierung der gefunden MR sowie die Generierung von Testfällen. Dies sind jedoch Themen für spätere Arbeiten.

4.3 Herausforderungen

In diesem Kapitel wollen wir die Probleme und Schwierigkeiten dokumentieren, welche während der Entwicklung des Programms aufgetreten sind. Möglicherweise können so Fehler bei ähnlichen Arbeiten vermieden und umgangen werden.

4.3.1 Fehlerhafte Auswertung der übersetzten Dateien

Während der Implementierung des Werkzeugs hat sich gezeigt, dass es problematisch ist, wenn die Reihenfolge, der vom Parser ausgegebenen Ausgabedaten, unterschiedlich ist. Aus diesem Grund ist es schwierig, die *Ausgabe Dateien* auf Textbasis zu vergleichen. In einer ersten Version unseres Programms fand der Vergleich der Ausgabedaten noch auf Textbasis statt. Konkret haben wir zu diesem Zeitpunkt die Ausgabedatei in mehrere Sektionen unterteilt, welche dann nacheinander auf leicht unterschiedliche Art und Weise verglichen wurden. Dabei haben wir unter anderem die Methode *unified_diff* aus dem Paket difflib[10] verwendet. Eine dieser Sektionen haben wir jedoch schon zu diesem Zeitpunkt auf JSON-Basis verglichen, da es in diesem Teil der Ausgabe immer zu Unterschieden in der Reihenfolge kam.
Da jedoch auch alle anderen Teile der Ausgabe von dieser Inkonsistenz in der Reihenfolge betroffen sind, ist es sinnvoll, auch die gesamte Ausgabe als JSON-Objekt einzulesen, zu sortieren und erst dann zu vergleichen.

[10] `https://docs.python.org/3/library/difflib.html`

4.3.2 Übersichtlichkeit

Bei der Entwicklung wurde schnell klar, dass unser Werkzeug relativ komplex und umfangreich werden wird. Daher war es sehr wichtig, den Code auf mehrere Python-Dateien aufzuteilen. Bevor wir angefangen haben, beispielsweise die Funktionen zur Wertegenerierung für die einzelnen Datentypen aufzuteilen, wurden die einzelnen Funktionen sehr schnell unübersichtlich. Auch jetzt ist der Programmcode unseres Programms noch immer nicht übersichtlich genug, daher sollte der Code in der Hinsicht noch verbessert werden.

4.3.3 Laufzeit

Ein weiteres Problem ist die Laufzeit. Der `exciting`-Parser benötigt nur wenige Sekunden, um eine Datei zu übersetzen. Da wir jedoch sehr viele *Eingabe Dateien* übersetzen müssen, steigt die Laufzeit des Programms schnell an. Dies hat es schwierig gemacht, unser Werkzeug im vollen Umfang zu testen. Während der Entwicklung sind wir daher so vorgegangen, dass wir das Programm meist nur auf wenigen Variablen haben laufen lassen.

5 Auswertung

Um unser Werkzeug zu bewerten, haben wir es mit mehreren *mainfiles* als *Standard-Eingabe* laufen lassen. Die *mainfiles* haben wir in zwei Gruppen aufgeteilt. Anschließend haben wir die Ergebnisse der beiden Gruppen verglichen. Dabei haben wir die Zahl der gefundenen MR betrachtet, aber auch geprüft, ob sich über die verschiedenen *mainfiles* hinweg gleiche MR finden lassen.

5.1 Aufbau des Experimentes

Wie bereits erwähnt haben wir zwei Gruppen von *mainfiles* gebildet. Jeder Gruppe sind 4 *mainfiles* zugeordnet. Die ersten beiden *mainfiles* sind Berechnungen, bei welchen Gallium(III) Oxide bzw. Kohlenstoff verwendet wurde. Wir haben hier zwei verschiedene Stoffe gewählt, um herauszufinden, ob sich bestimmte MR auch über die Materialien hinweg verifizieren lassen.
Die letzten beiden Dateien gehören zu Berechnungen, bei welchen das Material „Boron(III) Nitride“ betrachtet wurde. Während der Implementierung haben wir ein *mainfiles* als Referenz verwendet, dass eine Simulation mit dem Material „Boron(III) Nitride“ beschrieben hat.

In Tabelle 2 sind die von uns ausgewählten *mainfiles* noch einmal aufgeführt. Im git-Repositiry zu dieser Arbeit sind die vollständigen Links zu den Berechnungen, sowie alle Dateien enthalten, welche unser Programm während der Auswertung erzeugt und verwendet hat.

Gruppe 1

Nr	Material	Programmversion	Entry id
G1_F1	Gallium(III) Oxide	NITROGEN	znPcImrcW6PbToktDWG5MbMdo8_e
G1_F2	Carbon	OXYGEN	fL0MnyQ45rsaT8nxWRddgRiFvsFD
G1_F3	Boron(III) Nitride	NITROGEN-14	q5eFRvvLzpcNpXkEfp7neO8qr5ev
G1_F4	Boron(III) Nitride	NITROGEN-14	iGiyXyo4z2QzWVFpwJdGAi0QgCY3

Gruppe 2

Nr	Material	Programmversion	Entry id
G2_F1	Gallium(III) Oxide	NITROGEN	wDjCpU9CklzHMFHNIw1uYpOUGq8U
G2_F2	Carbon	OXYGEN	G2-ES2n20BlomnDt55yPVtb4L2x-
G2_F3	Boron(III) Nitride	NITROGEN-14	oJCWC5WHvKwwUgsPxxfxzn5x0CzK
G2_F4	Boron(III) Nitride	NITROGEN-14	k06u-7wXbzg1tcClFVGNLBtNgnUB

Tabelle 2: Tabelle: Verwendete Dateien für die Auswertung.

Name	Modell	CPU	Takt	RAM
gruenau[1-2]	Dell R740xd	Xeon 6254	3,1GHz	756GB
gruenau[5-8]	Dell R920	E7-4880	2,5GHz	1TB
gruenau9	Dell R740	Xeon 6134	3,2GHz	756GB
gruenau10	Dell R740xd	Xeon 6254	3,1GHz	756GB

Tabelle 3: Technische Daten der verwendeten Rechner.

5.2 Ausführung des Experimentes

Im ersten Schritt haben wir händisch die Variablen-Datei für jedes einzelne *mainfile* erstellt. Daraufhin haben wir unser Werkzeug mit den jeweiligen *mainfiles* gestartet und laufen lassen.

Um mehrere Berechnungen parallel auszuführen, haben wir unser Programm auf 8 verschiedenen Servern verwendet. Auf allen Server lief das Betriebssystem openSUSE Leap 15.3. Wir haben die Rechner gruenau1, gruenau2 und gruenau5-10 verwendet. Die technischen Daten zu den einzelnen Servern können Tabelle 3 entnommen werden. Weitere Informationen sind auf der Seite der Rechnerbetriebsgruppe [11] einsehbar.

5.3 Auswertung des Experimentes

In den folgenden Tabellen haben wir dargestellt, zu welchen Ergebnissen unser Werkzeug bei der Ausführung der 8 Dateien gekommen ist:

[11] `https://www.informatik.hu-berlin.de/de/org/rechnerbetriebsgruppe/dienste/hpc/computeserver`

Name	Steigende Werte		Fallende Werte	
	s. m. steigend	m. steigend	s. m. fallend	m. fallend
G1_F1	20	20	0	1
G1_F2	0	0	0	0
G1_F3	29	11	0	0
G1_F4	29	11	0	0
G2_F1	20	20	0	1
G2_F2	39	11	0	0
G2_F3	29	11	0	0
G2_F4	39	11	0	0

Tabelle 4: Anzahl der gefundenen MR auf Integer-Variablen.
Unterteilt in „monotone“ und „streng monotone“ Relationen.

Wichtig ist, in Tabelle 4 wird eine streng monoton steigende bzw. fallende MR nicht mehr als monoton steigende bzw. fallende MR mitgezählt.

Beim Vergleich der Tabellen wird schnell deutlich, dass für den Variablentyp Integer und Float deutlich mehr Relationen gefunden werden als für die anderen Datentypen. Das liegt vor allem daran, dass ein Großteil, der in den *mainfiles* vorkommenden Variablen, vom Typ Integer oder Float sind. Ebenso haben wir für Integer und Float die meisten Suchfunktionen für MR implementiert. Der Grund dafür ist, im Gegensatz zu MR in Strings, ist es einfacher MR auf Integer- oder Float-Daten zu finden.

Was auch deutlich ist, es werden in allen Dateien die gleiche Anzahl an MR für Strings gefunden. Abgesehen von G1_F2. Alle diese MR betreffen dabei dieselbe Variable. („Spin treatment“)

Was ebenfalls auffällt ist, dass bei der Datei G1_F2, abgesehen von zwei MR auf Strings, keine weiteren MR gefunden wurden. Der Grund hierfür ist, dass das verwendete *mainfile* wohl einen Fehler beinhaltet, denn alle Rechnungen brechen mit folgender Parser-Fehlermeldung ab:

```
ERROR    nomad.cli    2022-05-19T03:27:06
no "representative"section system found
```

Da schon das unveränderte *mainfile* diesen Fehler hervorruft, scheint es keinen Fehler diesbezüglich in unserem Werkzeug zu geben. Zur Berechnung von G1_F2 wurde die `exciting`-Version Oxygen verwendet. Jedoch wurde die Version Oxygen auch für das *mainfile* G2_F2 verwendet, wo es zu keinem solchen Fehler kam. Daher kann die Version von `exciting` hier nicht der Grund für den Fehler sein. Was der genaue Grund für diesen Fehler ist, können wir daher nicht sagen.

Um eine Aussage treffen zu können, ob sich die gefunden MR auf unseren Daten

Name	Identitäten			Negationen		
	f. alle W.	f. fast alle Werte	f. einzelne Werte	f. alle. W.	f. fast alle Werte	f. einzelne Werte
G1_F1	36	0	24	107	4	10
G1_F2	0	0	0	0	0	0
G1_F3	96	0	25	157	3	13
G1_F4	96	0	25	157	3	13
G2_F1	36	0	24	110	4	10
G2_F2	101	0	35	172	3	22
G2_F3	96	0	25	157	3	13
G2_F4	101	0	35	172	3	22

Tabelle 5: Anzahl der gefundenen MR auf Integer-Variablen.
Unterteilt in: Relation gilt „für alle Werte“, „für fast alle Werte“ und „für einzelne Werte“.

Name	Addition			Multiplikation		
	alle Werte	W. > 0	W. < 0	alle Werte	W. > 0	W. < 0
G1_F1	4	1	4	4	1	4
G1_F2	0	0	0	0	0	0
G1_F3	4	1	4	4	1	4
G1_F4	4	1	4	4	1	4
G2_F1	4	1	4	4	1	4
G2_F2	4	1	4	4	1	4
G2_F3	4	1	4	4	1	4
G2_F4	4	1	4	4	1	4

Tabelle 6: Anzahl der gefundenen MR auf Integer-Variablen.
Unterteilt in: Relation gilt „für alle Werte“, „für fast alle Werte größer als 0“ und „für alle Werte kleiner als 0“.

Name	Identität			Länge	
	f. alle Werte	f. fast alle Werte	f. einzelne Werte	f. alle Werte	f. fast alle Werte
G1_F1	2	0	3	4	0
G1_F2	0	0	0	0	0
G1_F3	2	0	2	4	0
G1_F4	2	0	2	4	0
G2_F1	2	0	3	4	0
G2_F2	2	0	2	4	0
G2_F3	2	0	2	4	0
G2_F4	2	0	2	4	0

Tabelle 7: Anzahl der gefundenen MR auf Listen (mit Elementen vom Typ Integer oder Float). Unterteilt in: Relation gilt „für alle Werte“, „für fast alle Werte“ und „für einzelne Werte“.

Name	Identität		
	f. alle W.	f. fast alle W.	f. einzelne W.
G1_F1	1	0	0
G1_F2	0	0	0
G1_F3	1	0	0
G1_F4	1	0	0
G2_F1	1	0	0
G2_F2	1	0	0
G2_F3	1	0	0
G2_F4	1	0	0

Tabelle 8: Anzahl der gefundenen MR auf Vektoren.
Unterteilt in: Relation gilt „für alle Werte“, „für fast alle Werte“ und „für einzelne Werte“.

Name	Identität			Leerzeichen		Andere Datentypen werden verarbeitet	
	f. alle Werte	f. fast alle Werte	f. einzelne Werte	möglich	nicht möglich	ja	nein
G1_F1	1	0	0	0	1	0	1
G1_F2	0	0	0	0	1	0	1
G1_F3	1	0	0	0	1	0	1
G1_F4	1	0	0	0	1	0	1
G2_F1	1	0	0	0	1	0	1
G2_F2	1	0	0	0	1	0	1
G2_F3	1	0	0	0	1	0	1
G2_F4	1	0	0	0	1	0	1

Tabelle 9: Anzahl der gefundenen MR auf Strings.
Unterteilt in: Relation gilt „für alle Werte“, „für fast alle Werte“ und „für einzelne Werte“.

verallgemeinern lassen, vergleichen wir die Dateien G1_F3 und G1_F4 sowie G2_F3 und G2_F4 miteinander. Alle vier *mainfiles* gehören zu Berechnungen mit dem Stoff Boron(III) Nitride.

Die gefundenen MR der beiden Dateien G1_F3 und G1_F4 unterscheiden sich lediglich bei den MR der Identität für einzelne Werte. Der Unterschied dort kommt durch die unterschiedlichen Ausgangswerte im *mainfile* zustande. Auch gibt es bei einigen MR Unterschiede in den angegebenen JSON-Pfaden, an welchen in der Ausgabe-Datei die Änderung gefunden wurden, jedoch können diese Unterschiede unserer Meinung nach vernachlässigt werden.

Bei den Dateien G2_F3 und G2_F4 gibt es mehr Unterschiede. Aus Tabelle 4 und Tabelle 5 ist bereits zu entnehmen, dass in G2_F4 mehr MR gefunden wurden als in G2_F3. Jedoch sind alle MR, welche in G2_F3 gefunden wurden, auch in G2_F4 vorhanden. Jedoch gibt es hier wieder die bereits erwähnten Unterschiede bei den MR für die Identität bei einzelnen Werten und den Pfaden mancher MR. Vergleichen wir nun G1_F3 und G2_F3 miteinander, so finden sich auch hier nur die bereits erwähnten Unterschiede. Daher können wir sagen, dass auf unseren Daten alle MR, die auf G1_F3, G1_F4 und G1_F3 gefunden wurden, allgemeingültig für Rechnungen mit dem Stoff Boron(III) Nitride sind. Warum in G2_F4 weitere MR gefunden wurden, können wir nicht erklären.

Vergleichen wir nun G1_F1 und G2_F1 miteinander so fällt auf, dass es auch hier

Integer/Float					
Identität	Steigende Werte	Fallende Werte	Negation	Addition	Multiplikation
28	36	0	78	9	9
Listen		Vektoren	String		
Identität	Länge	Identität	Identität	Leerzeichen	A. Datentyp
2	4	1	1	1	1

Tabelle 10: Anzahl verifizierbarer MR.
Für eine bessere Übersicht haben wir alle MR eines Typs zusammengerechnet.

nur Unterschiede bei den MR für einzelne Identitäten und bei manchen JSON-Pfaden gibt. Jedoch werden in G2_F1, wie auch aus Tabelle 5 entnommen werden kann, 3 MR mehr gefunden. Es sind MR welche sich auf die Negierung der Werte beziehen.

Der Vergleich der Dateien G1_F2 und G2_F2 liefert leider keine weiteren Informationen. Um nun eine Aussage über MR treffen zu können, welche materialübergreifend gelten, vergleichen wir nun G1_F1 mit allen anderen Dateien.

Den Vergleich zwischen G1_F2 überspringen wir an dieser Stelle. Auch betrachten wir die Relationen für einzelne Werte nicht, da sich diese generell unterscheiden. (Negation, Identität)
Im Vergleich zwischen G1_F1 und G1_F3 können wir 78 der 253 MR aus G1_F1 nicht verifizieren. Damit kommen wir auf 175 gemeinsame MR. In Tabelle 10 sind die Zahlen der verifizierbaren MR dargestellt.
Den Vergleich zwischen G1_F1 und G1_F4 überspringen wir an der Stelle ebenso, da wir bereits feststellen konnten, dass sich in G1_F3 und G1_F4 dieselben MR befinden, die für den Vergleich herangezogen werden würden.

Zwischen den Dateien G1_F1 und G2_F2 lassen sich 60 der 253 MR nicht verifizieren, jedoch finden wir 193 gemeinsame MR.

Den Vergleich zwischen G2_F3 und G2_F4 überspringen wir an dieser Stelle, da sich mit den beiden Dateien keine weiteren MR verifizieren lassen. (Alle MR aus G1_F3 sind auch in G2_F3 und G2_F4 vorhanden. In G2_F4 sind zwar noch weitere MR vorhanden, diese können jedoch auch keine weiteren MR verifizieren, da die MR in G1_F3, G1_F4 und G2_F3 nicht vorhanden sind.)

Letztendlich haben wir die gemeinsamen MR der Dateien G1_F1-G1_F3 und G1_-F1-G2_F2 verglichen und sind dabei auf 170 gemeinsame Relationen gekommen, welche als allgemeingültig auf unseren Daten bezeichnet werden können. In Tabelle 10 haben wir die Anzahl der für unsere Daten allgemeingültigen MR dargestellt. Die MR, welche gefunden werden, sind sehr einfach. Es sind genau die MR, welche wir bereits

beschrieben haben. Wenn unser Werkzeug nach komplexeren MR suchen soll, dann müssen noch weitere Funktionen implementiert werden. Was uns jedoch aufgefallen ist, immer wenn die Identität auf allen Werten gefunden wurde, dann wird auch immer die Negations-Relation gefunden. Dies ist zwar richtig, jedoch hilft in diesem Fall die Negations-Relation nicht weiter, da die Identität eine stärkere Aussagekraft hat.

5.4 Diskussion

Unser Werkzeug kann auf dem `exciting`-Parser MR finden. Wahrscheinlich auch solche, die materialübergreifend gültig sind. Die Qualität der bisher gefundenen MR ist zwar noch nicht sehr hoch, jedoch sollte sich dies ändern lassen, indem das Programm weiterentwickelt wird. Unser Programm kann einfache Relationen wie Identitäten oder „steigende Werte“ finden. Etwas komplexer sind die MR, welche die Addition/Multiplikation mit konstanten Werten beschreiben, welche unser Werkzeug auch finden kann. Bisher können wir aber keine komplexen MR finden, wie beispielsweise:

$$\text{Ausgabewert} = \text{Eingabewert} * \text{Eingabewert}$$

MR, welche die Beziehung zwischen mehreren Variablen beschreiben, kann unser Werkzug bisher noch nicht finden.

Dennoch ist es aus unserer Sicht machbar ein Programm zu entwickeln, welches automatisiert nach MR auf dem `exciting`-Parser sucht, wie unser Prototyp zeigt. Ob sich mit den gefundenen MR Testfälle generieren lassen, mit welchen der Parser getestet werden kann, muss noch überprüft werden. Dies wäre ein Thema für eine nachfolgende Arbeit. Auch die Frage, ob sich unser Programm auf Parser der anderen NOMAD-Codes anwenden lässt, muss noch geklärt werden. Dennoch sollte aus unserer Sicht die Idee des MT der NOMAD-Parser weiterverfolgt werden.

5.5 Zukünftige Arbeit

In diesem Kapitel wollen wir auf zukünftige Erweiterungsmöglichkeiten für unser Werkzeug und unseren Ansatz eingehen. Erst werden wir auf die Aufgaben eingehen, welche im direkten Zusammenhang mit unserem Werkzeug stehen, danach auf Aufgaben, welche die gesamte Arbeit betreffen.

5.5.1 Zukünftige Aufgaben - exciting parser testing tool

Da unser Werkzeug bisher nicht vollständig ist, gibt es noch Aufgaben, auf welche wir in diesem Abschnitt eingehen. Wir wollen unsere Ideen präsentieren, wie das Programm erweitert und verbessert werden könnte.

Überarbeitung: Durch die kurze Implementierungszeit ist es an erster Stelle wichtig, das gesamte Werkzeug zu überarbeiten und kleinere Fehler zu beseitigen, sowie das „logging“ zu überarbeiten.

Erweiterung der Suchfunkionen nach MR: Bisher sind noch nicht für alle Datentypen Funktionen implementiert, um nach MR zu suchen. Beispielsweise für Variablen, welche nur bestimmte vordefinierte Werte als Eingabe akzeptieren. Auch sollten die bereits implementierten Funktionen erweitert werden.

Eine Idee, die MR-Suchfunktionen zu erweitern, wäre es, eine Schnittstelle zu implementieren. Über diese Schnittelle könnten zusätzlich externe Funktionen zur Suche nach MR übergeben werden. Somit wäre die Erweiterung unseres Programms einfacher, da auf diesem der Programmcode unseres Programms nicht bearbeitet werden müsste.

Erweiterung der Eingabemöglichkeiten: Es wäre von Vorteil, wenn die vordefinierten Werte durch weitere Eingaben ergänzt werden könnten. Dies würde die Möglichkeit schaffen, gezielt nach MR zu suchen. Diese Ergänzung könnte über Parameter umgesetzt werden.

Validierung der MR: Da wir nur wenige Werte berechnen, auf welchen wir nach MR suchen, könnte es dazu kommen, dass unser Werkzeug ungültige MR findet. Mit einer nachgelagerten Validierung könnte man dem entgegenwirken. Für die Validierung könnten weitere Eingabe-Dateien generiert und ausgeführt werden. Anhand der neuen Werte könnte dann mit größerer Sicherheit ausgesagt werden, ob es sich tatsächlich um eine MR handelt.

5.5.2 Zukünftige Aufgaben - Allgemein

Auch wenn unser Programm MR findet, wissen wir noch nicht, ob sich aus diesen MR auch Testfälle entwerfen lassen, um die Parser zu testen. Dies wäre ein Thema für eine weitere Arbeit. Ebenso wäre es interessant zu wissen, ob sich unser Werkzeug auch auf die Parser anderer Nomad-Codes anwenden lässt.

5.6 Gültigkeit

Da unser Werkzeug in der Ausführung, wie wir es für die Auswertung verwendet haben, nicht dem Einfluss von Zufall ausgesetzt ist, sollten alle Testergebnisse nachgestellt werden können. Angenommen, es wird die gleiche Umgebung verwendet, wie es bei unserer Ausführung der Fall war.

Für die Auswertung haben wir unser Programm parallel auf mehreren Rechnern laufen lassen. Alle Informationen zu den verwendeten Rechnern können aus Tabelle 3 entnommen werden. Da unser Werkzeug, wie bereits erwähnt, deterministisch ist, sollte jeder der verwendeten Rechner zu demselben Ergebnis einer Berechnung kommen. Auch haben wir auf allen Rechnern dieselbe Version von nomad-lab verwendet. (siehe Tabelle 1).

Was wir jedoch nicht ausschließen können ist, dass Fehler in unserem Programm vorhanden sind. Diese könnten unsere Ergebnisse verfälschen, wenn beispielsweise durch

einen Fehler in einer Methode nicht existente MR gefunden werden. Auch könnten Fehler in der *Variablen-Info Datei* vorhanden sein, da diese Dateien händisch erstellt wurden. Wir haben jedoch bei der Implementierung sowie bei der Ausführung des Experimentes darauf geachtet, exakt zu arbeiten.

6 Zusammenfassung

Das Ziel unserer Arbeit war es zu untersuchen, ob es möglich ist, automatisiert nach MR auf dem `exciting`-Parser zu suchen. Diese MR könnten dafür verwendet werden, den Parser metamorph zu testen. Der Grund dafür ist, derzeit gibt es noch keine strukturierten Tests der NOMAD-Parser.

Im Laufe dieser Arbeit haben wir ein Werkzeug entwickelt, mit welchem MR auf dem `exciting`-Parser gesucht werden können. Dabei verwenden wir eine `exciting`-Berechnung als Grundlage. Auf dieser Grundlage generieren wir weitere Eingabe-Dateien. Anschließend verwenden wir den `exciting`-Parser, um alle generierten und auch die als Grundlage verwendete Datei zu übersetzen. Die Ausgabe des Parsers verarbeiten wir dann weiter, um Informationen zu einzelnen Variablen zu erhalten. Auf diesen Informationen suchen wir dann im Anschluss nach MR.

Die Implementierung des Programms haben wir in Abschnitt 4 beschrieben. Ebenso haben wir in Unterabschnitt 4.3 die Probleme und Schwierigkeiten dokumentiert, welche während der Entwicklung aufgetreten sind.

Um unser Programm zu testen, haben wir verschiedene `exciting`-Simulationen als *Standard-Eingabe* für unser Werkzeug verwendet und geprüft, welche MR sich finden lassen.

Wir konnten zeigen, dass unser Werkzeug MR finden konnte, die zumindest auf unseren Daten materialübergreifend gültig sind. Wir konnten auf unseren Daten 170 MR als allgemeingültig verifizieren. Jedoch können wir bisher nur relativ einfache MR finden.

Dennoch konnten wir mit dieser Arbeit zeigen, dass es möglich ist MR auf dem `exciting`-Parser zu suchen. Jedoch muss noch Arbeit investiert werden, um aus unseren MR Testfälle zu generieren um damit den `exciting`-Parser zu testen.

Literatur

[Bon18] Bonnie Hardin, Upulee Kanewala. Using semi-supervised learning for predicting metamorphic relations, 2018.

[Cla18] Claudia Draxl and Matthias Scheffler. Nomad: The fair concept for big-data-driven materials science, 2018.

[Dan09] Daniel Hook and Diane Kelly. Testing for trustworthiness in scientific software, 2009.

[Eds72] Edsger W. Dijkstra. The humble programmer, 1972.

[exc] exciting. [Online; aufgerufen am 27.02.2022].

[Fan15] Fang-Hsiang Su, Jonathan Bell, Christian Murphy, Gail Kaiser. Dynamic inference of likely metamorphic properties to support differential testing, 2015.

[Jaf18] Jafar Alzubi et al. Machine learning from theory to algorithms: An overview, 2018.

[M. 99] M. D. Ernst, J. Cockrell, W. G. Griswold, D. Notkin. Dynamically discovering likely program invariants to support program evolution, 1999.

[Mar97] Mark Dowson. The ariane 5 software failure, 1997.

[Mar16] Mark D. Wilkinson et al. The fair guiding principles for scientific data management and stewardship, 2016.

[Ser16] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortes. A survey on metamorphic testing, 2016.

[Ser17] Sergio Segura, Amador Durán, Javier Troya, Antonio Ruiz Cortés. A template–based approach to describing metamorphic relations, 2017.

[Ser19] Sergio Segura, Dave Towey, Zhi Quan Zhou, Tsong Yueh Chen. Metamorphic testing: Testing the untestable, 2019.

[T. 98] T. Y. Chen, S. C. Cheung, S. M. Yui. Metamorphic testing: A new approach for generating next test cases, 1998.

[Upu13a] Upulee Kanewala and James M. Bieman. Techniques for testing scientific programs without an oracle, 2013.

[Upu13b] Upulee Kanewala, James M. Bieman. Using machine learning techniques to detect metamorphic relations for programs without test oracles, 2013.

[Upu15] Upulee Kanewala, James M. Bieman, Asa Ben-Hur. Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels, 2015.

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 25. Mai 2022 .....