



Stephan Josef Frühwirth, BSc

A practical approach to the compilation of MINION constraint models into SMT-LIB2 models

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Institute of Softwaretechnology

Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Graz, May 2023

This document is set in Palatino, compiled with [pdfL^AT_EX2e](#) and [Biber](#).

The L^AT_EX template from Karl Voit is based on [KOMA script](#) and can be found online: <https://github.com/novoid/LaTeX-KOMA-template>

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Acknowledgement

I would first like to thank my thesis advisor *Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa*. He inspired my interest in this topic a few years ago with the Compiler Construction lecture. That is why I chose him as my supervisor. The meetings with him were always productive and informative. He provided me with optimal support and important tips in the preparation, practical implementation, and, finally, for the written work.

I would also like to express a big thanks to *Roxane Koitz-Hristov* and *Birgit Hofer*. In their lectures on Software Maintenance, Compiler Construction and Compiler Optimization, and especially as a student assistant in their design practicals, I was able to build up much knowledge that has been incorporated into this thesis.

Special thanks go to my friend since my school days, *Matthias Müller-Murko*. He has accompanied and motivated me throughout my whole studies. He also proofread this thesis and gave valuable tips for improvement.

My very profound gratitude goes to *my family*, who have supported me throughout my life and made this study and my entire education possible in the first place. My father in particular, who asked me almost weekly when I would finish my studies, provided a periodic boost of motivation.

The biggest thanks, however, go to my dear wife, *Tatiana*, who stood by me through all the ups and downs during my studies' last and most crucial phase that lasted a surprisingly long time. Furthermore, she provided many valuable tips on writing scientific papers and thus contributed significantly to improving the quality of this thesis.

Abstract

Compilers are great tools for translating source code written in one programming language into another. This allows the execution of programs on different platforms, utilizing features of one problem-solving technique in other problem-solving systems, optimizing computer architectures, and more. One concrete application of compilers is the translation of Constraint Satisfiability Problems (CSP) into Boolean Satisfiability Problems (SAT) or Satisfiability Modulo Theories (SMT). In this master's thesis, we present a new compiler called MIN2SMT, written in Python and ANTLR. It compiles MINION Input Language to the SMT-LIB2 language. The MINION Input Language is used to model CSPs, and the SMT-LIB2 language is used to express logical formulas and constraints in SMT solvers. The compiler aims to provide a more accessible way of defining and solving optimization problems using the simple syntax of MINION and to integrate these problems into larger systems that use SMT solvers. In order to be able to check the correctness of the translation, the term *equivalence* must be defined. A test framework explicitly developed for MIN2SMT uses a large number of test files to demonstrate this equivalence. Optimizations of the compiler ultimately lead to performance improvements of the generated code, thus forming the compiler into a good tool that can translate CSPs in SMT.

Kurzfassung

Compiler sind großartige Werkzeuge für die Übersetzung von einer Programmiersprache in eine andere. Das ermöglicht die Ausführung von Programmen auf verschiedenen Plattformen, die Nutzung von Eigenschaften einer Problemlösungstechnik in anderen Problemlösungssystemen, die Optimierung von Computerarchitekturen und vieles mehr. Eine konkrete Anwendung von Compilern ist die Übersetzung von Constraint Satisfiability Problems (CSP) in Boolean Satisfiability Probleme (SAT) oder Satisfiability Modulo Theories (SMT). In dieser Masterarbeit stellen wir einen neuen Compiler namens MIN2SMT vor, der in Python und ANTLR geschrieben wurde. Er kompiliert die MINION Input Language in SMT-LIB2. Die MINION Input Language wird verwendet, um CSPs zu modellieren, SMT-LIB2 wird hingegen verwendet, um logische Formeln und Constraints in SMT Solvern auszudrücken. Der Compiler zielt darauf ab, einen leichter zugänglichen Weg zur Definition und Lösung von Optimierungsproblemen unter Verwendung der einfachen Syntax von MINION zu bieten und diese Probleme in größere Systeme zu integrieren, die SMT Solver verwenden. Um die Korrektheit der Übersetzung überprüfen zu können, muss der Begriff *Äquivalenz* definiert werden. Ein explizit für MIN2SMT entwickeltes Testframework nutzt eine Vielzahl von Testdateien, um diese Äquivalenz nachzuweisen. Diverse Optimierungen des Compilers führen letztlich zu Leistungsverbesserungen des generierten Codes, die den Compiler zu einem guten Werkzeug machen, das CSPs in SMT übersetzen kann.

Contents

Abstract	v
Kurzfassung	v
1. Introduction	1
2. Solver	4
2.1. Constraint Solver	4
2.1.1. Applications of constraint solvers	6
2.1.2. MINION	6
2.2. Satisfiability	7
2.3. Satisfiability Modulo Theories	9
2.3.1. SMT Solver	11
2.3.2. SMT-LIB2	11
2.3.3. Z ₃	11
3. Compiler	14
3.1. Lexer	14
3.2. Parser	15
3.3. Intermediate code	18
3.4. ANTLR	20
4. Compilation of MINION to SMT-LIB2	23
4.1. Implementation	23
4.1.1. Used libraries	24
4.2. Architecture of the compiler	24
4.3. Translation of statements	31
4.4. Limitations	54

5. Testing, optimizations, and performance	56
5.1. Unit testing	56
5.2. Integration testing	57
5.2.1. Test data	60
5.3. Optimizations	61
5.3.1. Removal of unused variables	61
5.3.2. Geq/Leq Optimization	62
5.3.3. Sum Constraints	62
5.3.4. Table Constraints	65
5.4. Performance discussion	68
5.4.1. Geq/Leq Optimization	68
5.4.2. Sum Constraints	69
5.4.3. Table Constraints	70
5.4.4. Summary	70
6. Conclusion and future work	75
7. Abbreviations	78
A. Supplementary code listings	80
Bibliography	84

List of Figures

1.1. The compiler must generate semantically equivalent translations.	2
3.1. A parse tree representing the Listing 3.2	18
3.2. A CFG constructed from Listing 3.4	19
3.3. Workflow of implementing a compiler with the help of ANTLR	21
4.1. A class diagram of the arguments class and its derived child classes.	28
4.2. A class diagram of the Smtlib2Statement class and its derived child classes.	29
5.1. Workflow of the test framework in the UNSAT case.	58
5.2. Workflow of the test framework in the SAT case.	59

1. Introduction

Compilers play a crucial role in computer science as they enable the translation of source code written in one programming language into another, allowing the execution of programs on different architectures and platforms. The process of compiler development involves designing and implementing a set of algorithms and techniques to generate efficient and correct code. In recent years, there has been a growing interest in developing compilers using modern programming languages, such as Python, to gain the ability to write programs in simpler languages and translate them to more complex languages that can solve the original problem more efficiently.

This thesis presents a compiler, [MIN2SMT](#) [[Frü23](#)], written in Python and ANTLR4 which compiles the MINION Input Language [[Jef+14](#)] to the [SMT-LIB2](#) language [[GJM06](#)]. The MINION Input Language allows users to model Constraint Satisfiability Problems (CSP). The [SMT-LIB2](#) language ([[Ini23](#)]), on the contrary, is used to express logical formulas and constraints in Satisfiability Modulo Theories (SMT) solvers.

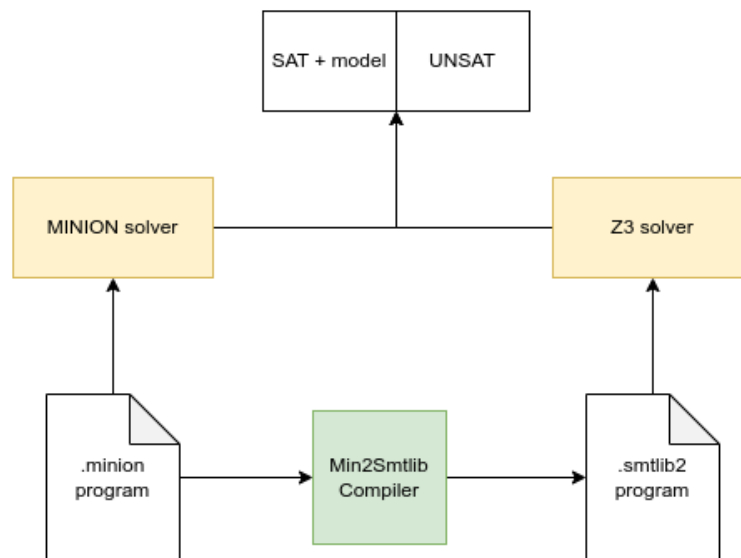
Franch [[Fra12](#)] figured out that there already exist some promising solutions for translating Constraint Satisfiability Problems into Boolean Satisfiability Problems (SAT). However, no tool was found that translates CSPs into SMT until Bofill, Suy, and Villaret [[BSV10](#)] developed Simply, a declarative programming system for easy modeling and solving of CSPs. Currently, MINION was used in various projects and experiments at the Institute of Software Technology, Graz University of Technology, such as [[Hof14](#); [Hof+](#); [WNA10](#)]. Nevertheless, the development of a tool for the translation of MINION constraints into SMT models - [SMT-LIB2](#) models, to be more precise - is still pending. Therefore, [MIN2SMT](#) was developed in the course of this master's thesis.

1. Introduction

The goal of this compiler is to provide a more accessible way of defining and solving optimization problems using MINION, as well as to enable the integration of MINION into larger systems that use SMT solvers. The compiler takes MINION programs as input and generates SMT-LIB2 code as output which can be executed using the Z3 SMT solver ([Res23a]) that supports the SMT-LIB2 language.

The ANTLR4 parser generator generates a lexer and parser for the MINION Input Language, which can be easily extended to support new language constructs. Implementing the compiler involves several stages: lexical analysis, parsing, semantic analysis, and code generation. The Python programming language is used to implement the semantic analysis and code generation stages which involve translating the abstract syntax tree (AST) produced by the parser into SMT-LIB2 code. The result, a .smtlib2 file, must be *semantically equivalent* to the original file. In other words: if MINION finds a solution that satisfies all constraints, this solution must also satisfy the translated constraints. On the other side, if MINION returns UNSAT on a given input, Z3 must also return UNSAT on the translated file. Therefore, a test framework had to be developed to check the automatically generated files for *equivalence*.

Figure 1.1.: The compiler must generate semantically equivalent translations.



The remainder of this thesis is structured as follows. [Chapter 2](#) provides an overview of constraint solvers, Boolean Satisfiability and SMT-solvers, with a detailed discussion of the MINION constraint solver and the [SMT-LIB2](#) language. [Chapter 3](#) outlines the structure and tasks of the compiler. In [chapter 4](#), the design and implementation of the [MIN2SMT](#) compiler are presented, including the translation of MINION statements to [SMT-LIB2](#) statements. [Chapter 5](#) covers the testing methods, data, and framework used to demonstrate the equivalence of the compiler's output. Furthermore, the implemented optimization techniques will be presented in this chapter. Finally, [chapter 6](#) concludes the thesis and discusses possible directions for future work.

2. Solver

Constraint solvers, *Boolean Satisfiability (SAT)* solvers and *Satisfiability Modulo Theories (SMT)* solvers are powerful tools that have revolutionized the field of automated reasoning and decision-making. They are widely used in various domains to solve complex problems involving constraints and logical formulas. Furthermore, as Höfler [Höf15] points out, these technologies' underlying ideas are identical, and the fields of application for these solvers are closely interconnected.

All three concepts provide powerful algorithms for solving complex problems that are difficult or impossible to solve using traditional methods. They are widely used in academic research and industrial applications and have contributed significantly to advancing the state-of-the-art in areas such as artificial intelligence, robotics, and software engineering. [DGN10; Rya10; Simo1]

This chapter will explore the background of constraint solvers and SMT solvers. Furthermore, in order to understand SMT solvers, the concept of Boolean Satisfiability and SAT solvers are also briefly addressed. In the course of this chapter, the key features and applications of the different solver concepts will be discussed.

2.1. Constraint Solver

A constraint solver is a tool for solving complex problems in fields such as artificial intelligence, software validation and verification, and computer science in general. The goal of a constraint solver is to find an *assignment* or *model* for every variable that satisfies a set of constraints. Such satisfiability problems are also called Constraint Satisfiability Problems (CSP). [Höf15]

Brailsford, Potts, and Smith [BPS99] define a **CSP** as follows:

- a set of variables X_1, X_2, \dots, X_n ;
- a set of possible values D_1, D_2, \dots, D_n for each variable, the so called domain;
- a set of constraints that can be seen as relations between variables. Constraints restrict the values that the variables can have at the same time.

If an assignment to every variable X_i with a value from its domain D_i can be found, then a solution to a **CSP** is found, and therefore, the **CSP** is satisfiable. In contrast, if no solution can be found to satisfy all constraints, the **CSP** is unsatisfiable.

There are many different types of constraints that a constraint solver can handle. The most common types include:

- *Arithmetic constraints*: These constraints involve arithmetic operations, such as addition, subtraction, multiplication, and division. For example, a constraint solver might be used to solve an optimization problem that involves maximizing a function subject to a set of arithmetic constraints.
- *Boolean constraints* involve logical operations, such as AND, OR, NOT, XOR, and IMPLICATION.

One of the key benefits of a constraint solver is that it can handle complex, real-world problems with many variables and constraints. For example, a constraint solver might be used to optimize a complex manufacturing process that involves multiple stages, resources, and constraints, as the *vehicle routing problem* described in [BPS99]. Such problems might be too difficult to solve without a constraint solver using traditional optimization techniques.

In addition to its power and flexibility, a constraint solver has some limitations. One major limitation is that it can be computationally expensive, especially for significant problems with many variables and constraints, as demonstrated in [chapter 5](#). In addition, the problem definition may become complex due to the limited number of available constraints.

Although there are some limitations, constraint solvers are an invaluable tool for solving complex problems. With the continuous advancement of technology, constraint solvers will become even more powerful and extensively utilized across an even more comprehensive range of applications.

2.1.1. Applications of constraint solvers

Constraint solvers can be used to solve a wide range of problems. Besides others, Höfler [Höf15] and Simonis [Sim01] mention resource planning, distribution planning, bank network balancing and network traffic minimization, production planning and scheduling, transport planning, and personnel allocation as examples for the usage of constraint programming in industrial applications. Brailsford, Potts, and Smith [BPS99] further supplement this list with the cutting stock problem, the car sequencing problem, and the location of facilities.

Listing 2.1 demonstrates a simple CSP: *The farmers problem*. Jefferson et al. [Jef+14] describes this problem as follows:

"A farmer has seven animals on his farm: pigs and hens. They all together have 22 legs. How many pigs (4 legs) and how many hens (2 legs) does the farmer have?"

2.1.2. MINION

MINION is a fast and scalable constraint solver. As Gent, Jefferson, and Miguel [GJM06] point out, their goal was to build an easy-to-use constraint solver with a syntax that is as simple as possible but simultaneously solves problems quickly and efficiently. With this approach, they want to support the *'model and run'* paradigm promoted by Puget [Pugo4]. MINION is also intended to be a counterpart to constraint toolkits. Constraint toolkits are libraries for various programming languages such as C++, Java, or Prolog. Their disadvantage, however, is that constraint solving of larger problems requires a high degree of experience and in-depth knowledge of the often complex internal architecture of the toolkits.

Listing 2.1: *The farmers problem: An example of a CSP written in MINION.*

```
1 MINION 3
2 **VARIABLES**
3 DISCRETE pigs {0..7}
4 DISCRETE hens {0..7}
5
6 **CONSTRAINTS**
7 weightedsumgeq([2,4], [hens, pigs], 22)
8 weightedsumleq([2,4], [hens, pigs], 22)
9 sumleq([hens,pigs], 7)
10 sumgeq([hens,pigs], 7)
11
12 **EOF**
```

MINION has four types of variables:

- binary variables, so-called 0/1 variables;
- bound variables;
- sparse bound variables;
- discrete variables.

This constraint solver supports a wide variety of constraints. The syntax of the MINION constraints is similar to that of function calls of various high-level languages such as C. Besides constraints for modeling Boolean properties such as equality, disequality, and inequality, also constraints for modeling arithmetical problems like sum or product are provided. In addition, constraints are available for the description of tables. Another important concept that is supported is *reification*, to assign the satisfaction or unsatisfaction of a constraint to a Boolean variable. [GJMo6]

2.2. Satisfiability

The problem of Boolean Satisfiability (**SAT**) involves determining whether it is possible to assign values to the variables in a given Boolean formula,

such that the formula evaluates to *true*. Such formulas are denoted *satisfiable* (**SAT**). On the contrary, another important task is to determine if no such assignment is possible, which would indicate that the function expressed by the formula is *unsatisfiable* (**UNSAT**) for any possible variable assignment. [Fra12]

As Höfler [Höf15] explains, the **SAT** problem was the first to be demonstrated as NP-Complete. As a result, no known algorithms can solve all **SAT** instances in a reasonable amount of time. Despite this limitation, the importance of **SAT** solving for a variety of applications has made it an upcoming research area over the last decades. Thanks to the progress made in modern **SAT** solvers, they are now capable of efficiently solving many challenging real-world problems across various domains.

SAT solvers operate on the language of propositional logic. This language consists of

- Boolean variables;
- Boolean operators negation (\neg), conjunction (\wedge) and disjunction (\vee).

As a consequence, propositional formulas for Boolean Satisfiability are composed of Boolean variables connected with relations of Boolean algebra. [Elg15]

The following examples give an idea of propositional formulas:

$$\neg A \tag{2.1}$$

$$(A \vee \neg B \vee C) \wedge \neg A \tag{2.2}$$

$$\neg A \wedge (B \vee C) \tag{2.3}$$

$$A \vee B \tag{2.4}$$

where A , B and C are Boolean variables.

Propositional formulas can be used to represent a series of *declarative sentences*. A declarative sentence is a sentence (statement) that is either *true* or *false*. They can consist of only one propositional atom, like *The sun is shining*, or have a more complex structure - *If today is Monday, tomorrow is Tuesday*. An example for a non-declarative sentence would be *What time is it?*, since this is not a statement that can be *true* or *false*. Other examples of such non-declarative sentences would be exclamations or commands. [Hof15]

[SAT](#) solving has proven to be a valuable tool for numerous applications. In fact, [SAT](#) solvers serve as a fundamental component for certain digital circuit design applications in use today. Elgabou [[Elg15](#)] lists applications such as software and hardware testing, model checking, design verification, and debugging.

A notable distinction between propositional satisfiability problems and Constraint Satisfiability Problems is that [SAT](#) problems feature binary domains and non-binary constraints, whereas [CSP](#)s typically involve non-binary domains and binary constraints. However, it is possible to translate any [SAT](#) problem into a [CSP](#) and vice versa in polynomial time since both problem classes are NP-complete. [[Waloo](#)]

2.3. Satisfiability Modulo Theories

Satisfiability Modulo Theories are closely related to Boolean Satisfiability ([SAT](#)), as [SMT](#) can be seen as extension to [SAT](#). The majority of [SMT](#) solvers use [SAT](#) solvers as a basis to check whether an [SMT](#) problem can be solved.

As mentioned above, [SMT](#) uses propositional logic and extends it by first-order logic. Therefore, a [SMT](#) problem can be defined as determining whether a formula in first-order logic is satisfiable or not. Höfler [[Höf15](#)] and Moura and Bjørner [[MBo9](#)] define the additional elements used by [SMT](#) as follows:

- *Variables* of different types, such as Boolean, integer or real [[Dav13](#)].
- *Truth constants*: *true* and *false*.
- *Operators*: In addition to the conjunction, disjunction, and negation, more operators are supported, in detail: the *equality* operator ($=$) and the binary connective *implication* (\Rightarrow).
- *Predicates* (*predicate symbols* or *relation symbols*): Predicates take one or more arguments and can be *true* or *false*, depending on the argument(s) of the predicate. Simple predicates with no arguments are equivalent to Boolean variables. In addition, predicates also include relational operators, such as $>$, $<$, \geq , and \leq .

- *Functions*: functions take one or more arguments and return a value. Functions with no arguments are equal to constants.
- *Quantifiers*: the *exists* (\exists) and *forall* (\forall) quantifiers with bound variables. Let P be a formula and x is a variable, then $\forall x P(x)$ means that for all x , P holds and $\exists x P(x)$ means that there exists x such that P holds.

SMT further introduces the concept of *theories*. A theory is a set of sentences, where a *sentence* is a formula without free variables. Generally, a Σ -theory is a collection of sentences over a signature Σ . A formula φ is satisfiable modulo a theory T if $T \cup \{\varphi\}$ is satisfiable. This means that there exists a model M that satisfies φ using the theory T , also denoted as $M \models_T \varphi$. [Höf15; MB09]

As Moura and Bjørner [MB09] point out, there exists a huge variety of different theories, such as

- Linear Arithmetic;
- Difference Arithmetic;
- Non-linear Arithmetic;
- Free Functions (also known as *Theory of uninterpreted functions*);
- Bit-Vectors;
- Arrays;
- Different data types, i.e. *strings*, *lists*, *pairs*.

These theories can be used individually but also in combination with other theories, for instance, linear algebra expressed via arrays. Due to a large number of different theories available and the additional possibilities that the use of first-order logic brings, **SMT** problems can be formulated more quickly and efficiently compared to **CSP** and **SAT**. However, **CSP** constraints and **SAT** models can be translated into **SMT** models to take advantage of **SMT** or to embed their concepts into a larger **SMT** framework, as was done in the course of the development of **MIN2SMT**.

Elgabou [Elg15] gives multiple examples for **SMT** applications, such as software testing and verification, model checking, and theorem proving. Moreover, **SMT** solvers such as **Z3** can be further used for model-based software development, compiler validation, network verification, and optimization. [Res23a]

2.3.1. SMT Solver

Satisfiability Modulo Theories ([SMT](#)) solvers are automated reasoning tools used in computer science and engineering to solve logical formulas involving multiple theories. An [SMT](#) solver can decide the satisfiability of a formula consisting of a combination of first-order logic with different theories such as arithmetic, bit vectors, arrays, and uninterpreted functions. The main goal of an [SMT](#) solver is to find an assignment of values to the variables in the formula that satisfies the formula.

[SMT](#) solvers are widely used in various applications such as software verification, program synthesis, optimization, and security analysis. They can also be used for solving problems that involve a combination of different theories, which are challenging to solve using traditional theorem-proving techniques.

2.3.2. SMT-LIB2

[SMT-LIB2](#) (Satisfiability Modulo Theories Library, version 2) is a standardized input language and format for [SMT](#) solvers. It was developed by the [SMT-LIB](#) community to provide a common interface for [SMT](#) solvers and facilitate the development of new solvers and applications that use [SMT](#) technology. [[Ini23](#)]

The [SMT-LIB2](#) language provides a way to express logical formulas in a standard format that multiple solvers can understand. This allows users to write a single formula and test it against multiple solvers, making it easier to compare the performance and results of different solvers. [Listing 2.2](#) demonstrates how the example of *The farmers problem* in [listing 2.1](#) can be translated to [SMT-LIB2](#).

2.3.3. Z3

One of the most popular [SMT](#) solvers that support [SMT-LIB2](#) is the [Z3](#) solver. Developed by Microsoft Research, [Z3](#) is a high-performance [SMT](#) solver that

Listing 2.2: "The farmers problem", the original CSP written in MINION and translated to SMT-LIB2.

```
1 (declare-const pigs Int)
2 (declare-const hens Int)
3
4 ; pigs {0..7}
5 (assert (<= 0 pigs 7))
6 ; hens {0..7}
7 (assert (<= 0 hens 7))
8 ; weightedsumeq {custom}([2, 4], [hens, pigs], 22)
9 (assert (= (+ (* 2 hens) (* 4 pigs)) 22))
10 ; sumeq {custom}([hens, pigs], 7)
11 (assert (= (+ hens pigs) 7))
12
13 (check-sat)
```

supports a wide range of theories, including arithmetic, arrays, bit-vectors, and quantifiers. [Res23b]

As [Höf15] points out, [Z3](#) is known for its speed and scalability and is widely used in academic research and industrial applications. It has been used to solve various problems, including software verification, program analysis, and optimization.

One of the key features of [Z3](#) is its ability to handle complex formulas and theories. It can solve formulas that involve multiple theories and as well as handle quantifiers that are often used in program verification and optimization.

As [Res23b] states, [Z3](#) also provides an API that allows users to integrate this solver into their applications and tools and extend [Z3](#)'s functionality to meet their specific needs. This API provides a way to interact with [Z3](#) from other programming languages, such as Python, Java, and C++.

In summary, [SMT-LIB2](#) and the [Z3](#) solver provide a powerful toolset for solving complex problems that involve logical formulas and constraints. The standardization of the [SMT-LIB2](#) language allows for easier comparison and

integration of different solvers, while the performance and flexibility of the Z_3 solver make it a popular choice for a wide range of applications which was also shown by Höfler [Höf15] in an extensive comparison of different SMT solvers, including Z_3 .

3. Compiler

Compilers play a critical role in modern computer science and software engineering, enabling programmers to write high-level code that is more human-readable and intuitive while optimizing performance and ensuring program correctness. They accomplish this by translating programming languages, such as Java or C++, into machine-readable code that can be executed by a computer's central processing unit (CPU). Furthermore, compilers are used to translate easy-to-read languages such as MINION into more complex languages such as [SMT-LIB2](#). As [\[Aho+06\]](#) lines out, we can distinguish the following applications of compiler technology:

- Implementation of high-level programming languages;
- Optimization for computer architectures;
- Design of new computer architectures;
- Program translations;
- Software productivity tools.

This chapter will explore the inner workings of compilers, including the various stages involved in the compilation process. In the course of this, the lexer, parser, and intermediate code representation as typical parts of a compiler will be discussed. Furthermore, a parser generator called [ANTLR](#) will be presented, which was used for developing the [MIN2SMT](#) compiler.

3.1. Lexer

A lexer, also known as a lexical analyzer or scanner, is the first part of a compiler that performs the initial processing of a source code file to produce a stream of tokens that a parser can further process. As Koitz-Hristov et al. [\[Koi+19\]](#) pointed out, the lexer is a crucial component in most compilers

and interpreters, and its primary function is to split the input text into individual tokens, which are the basic building blocks of the language being processed. A token is composed of a token name and an attribute value.

The process of lexical analysis involves examining the input text and recognizing the individual lexemes that make up the language. A lexeme is a sequence of characters representing a pattern for a token. For example, in the C programming language, the sequence of characters *int* represents the token for an integer type. The lexer's job is to recognize these sequences of characters and convert them into a stream of tokens that can be passed on to the parser. [Aho+06]

Koitz-Hristov et al. [Koi+19] further explain that the lexer operates by reading characters from the input text and matching them against a set of predefined patterns or regular expressions that define the lexemes of the language. The lexer then continues to read characters and match them against patterns until the entire input file has been processed. The corresponding token is generated and added to the output stream when a match is found.

In addition to generating tokens, the lexer may perform other tasks, such as removing comments and whitespace from the input text. This helps to simplify the input further and makes it easier for the parser to process.

In conclusion, the lexer is a critical component of the compiler or interpreter, as it is responsible for breaking down the input text into smaller chunks and preparing it for further processing by the parser.

3.2. Parser

A parser acts as a syntactic analyzer. It is a program component that takes the stream of tokens generated by the lexer and constructs a tree-like structure that represents the syntactic structure of the input text. The parser is responsible for enforcing the rules of the language's grammar and ensuring that the input conforms to the language's syntax.

3. Compiler

Listing 3.1: A subset of lexer rules for the MINION language definition, written for ANTLR

```
1 WS : [ \r\n\t ] + -> channel (HIDDEN);
2
3 COMMENT : '#' ~[\n\r]* -> channel(HIDDEN);
4
5 fragment NON_ZERO_DIGIT: '1'..'9';
6 fragment DIGIT: '0'..'9';
7 fragment LETTER : LOWERCASE | UPPERCASE;
8 fragment UPPERCASE : 'A'..'Z';
9 fragment LOWERCASE : 'a'..'z';
10
11 INT : '0' | '-'? NON_ZERO_DIGIT DIGIT*;
12 ID : (LETTER | DIGIT | '_' | '-')+;
```

The parser works by analyzing the sequence of tokens and using a set of rules that define the language's syntax to construct a parse tree. As Aho et al. [Aho+06] point out, an abstract syntax tree (AST) is a data structure used to represent the structure of source code in a programming language. It is a tree-like structure that captures the essential elements of the program's syntax, ignoring the details of the source code's formatting and layout. Each node in the tree corresponds to a syntactic element of the program, such as a variable, function call, or control flow construct. The tree's internal structure reflects the hierarchy of the program's syntax, with each node representing a syntactic construct that contains other nodes. Figure 3.1 demonstrates a simple parse tree generated from listing 3.2.

Once the parser has constructed the parse tree, it may perform additional checks to ensure that the input is semantically valid. This task may involve checking for type errors, verifying that variables are defined before use, or enforcing other constraints of the language's semantics.

After the creation of the AST has been completed, and further semantic checks have been carried out, the intermediate code can be created utilizing the AST data structure.

3. Compiler

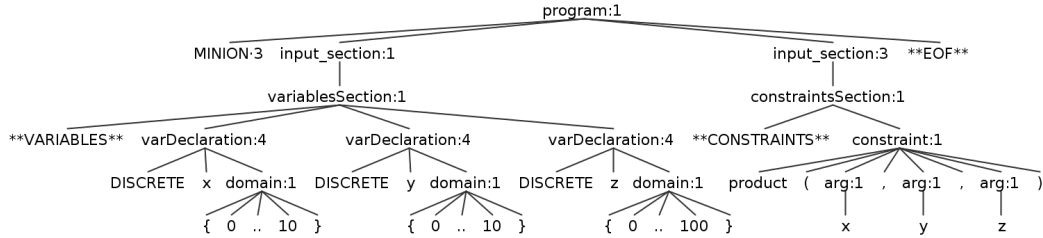
Listing 3.2: A simple MINION program which ensures that $x \cdot y = z$

```
1 MINION 3
2
3 **VARIABLES**
4 DISCRETE x {0..10}
5 DISCRETE y {0..10}
6 DISCRETE z {0..100}
7
8 **CONSTRAINTS**
9
10 product(x, y, z)
11
12 **EOF**
```

Listing 3.3: A subset of parser rules for the MINION language definition.

```
1 program : KEY_MINION input_section+ KEY_EOF;
2
3 input_section :
4     variablesSection
5     | searchSection
6     | constraintsSection
7     | tuplelistSection;
8 variablesSection: KEY_VARIABLES varDeclaration*;
9
10 domain: '{' INT '..' INT '}';
11 domainSparsebound: '{' INT (',' INT)* '}';
12
13 listDeclaration: '[' INT (',' INT)* ']';
14
15 varDeclaration: KEY_BOOL ID listDeclaration?
16     | KEY_SPARSEBOUND ID listDeclaration? domainSparsebound
17     | KEY_BOUND ID listDeclaration? domain
18     | KEY_DISCRETE ID listDeclaration? domain
19     ;
```

Figure 3.1.: A parse tree representing the Listing 3.2



3.3. Intermediate code

Intermediate representation, or intermediate code, is a language-independent (at the back end), platform-independent (at the front end) low-level representation of a program used by compilers or interpreters to perform optimizations and generate executable code.

The purpose of intermediate code is to provide a standard format for the program that can be optimized and transformed to generate efficient machine code. Intermediate code is often designed to be easily analyzed and transformed, allowing compilers to perform optimizations that improve the performance of the generated code.

Intermediate code can be represented in various forms, including abstract syntax trees (ASTs), control flow graphs (CFGs), and three-address code (TAC). These representations provide different levels of abstraction and can be used for different purposes. Hofer described the different representations and their applications in [Hof18] as follows:

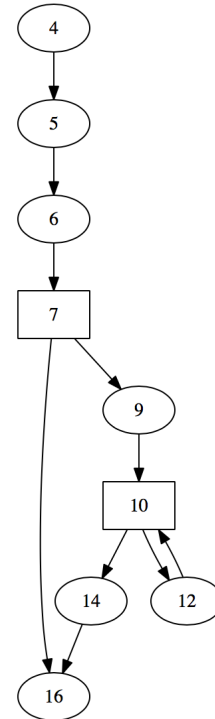
3. Compiler

- **ASTs** represent the program's structure as a tree, with each node in the tree representing a construct in the program, such as a statement or expression. **ASTs** can be used to perform optimizations such as constant folding and common subexpression elimination.
- **CFGs** represent the program's control flow as a directed graph, with each node representing a basic block of code and each edge representing a possible transfer of control. An edge on the **CFG** from a node n to a node n' only exists if and only if the instruction of n' can be executed immediately after the instruction of n . **CFGs** can be used to perform optimizations such as dead code elimination and loop optimization.

Listing 3.4: An exemplary Java code.

```
1 class TestClass {  
2     public static void  
3         main() {  
4         int x = 1;  
5         int y = 2;  
6         int z = 0;  
7         if (y > 0)  
8         {  
9             j = (4 + x);  
10            while(y > 0)  
11            {  
12                y = x + 1;  
13            }  
14            i = x;  
15            z = (z + y);  
16        }  
17    }
```

Figure 3.2.: A CFG constructed from Listing 3.4



- **TAC** represents expressions and statements as a sequence of operations that use a maximum of three operands. **TAC** is often used as the intermediate representation in compilers that generate code for register-based architectures. Furthermore, in [Koi+19] **TAC** was used to translate input code, similar to Java language, into code executed on

the Java Virtual Machine. Hofer [Hof18] gave the following example for TAC: $x := y \text{ op } z$, where x, y, z are arbitrary numbers, constants, names (variables), or temporary variables and op is an operator.

The choice of intermediate code representation depends on the goals of the compiler or interpreter and the target architecture. Some compilers may use multiple intermediate representations to perform different types of optimizations.

Intermediate code can also be helpful for debugging and analysis, as it provides a standardized format for the program that can be analyzed without knowledge of the source language. This can be useful for detecting security vulnerabilities or other issues that may be difficult to detect at the source code level. [Aho+06]

In the end, the intermediate code is then translated to the target language using the code generator.

3.4. ANTLR

ANTLR (ANother Tool for Language Recognition) [Par23a] is a powerful parser generator that can be used to build parsers, interpreters, and compilers for various programming languages and file formats.

ANTLR works by generating a parser from a formal grammar specification, written in its own syntax, which defines the structure and syntax of the language being parsed. The generated parser can then process input files written in the target language and perform actions based on the parsed input.

One of the critical benefits of ANTLR is its ability to generate parsers that can handle complex grammars, including ambiguous and left-recursive grammars. This makes it a valuable tool for developing compilers and interpreters for programming languages with complex syntax.

Another feature of ANTLR is its support for lexer rules which can be used to define the lexical structure of the parsed language. This allows ANTLR

3. Compiler

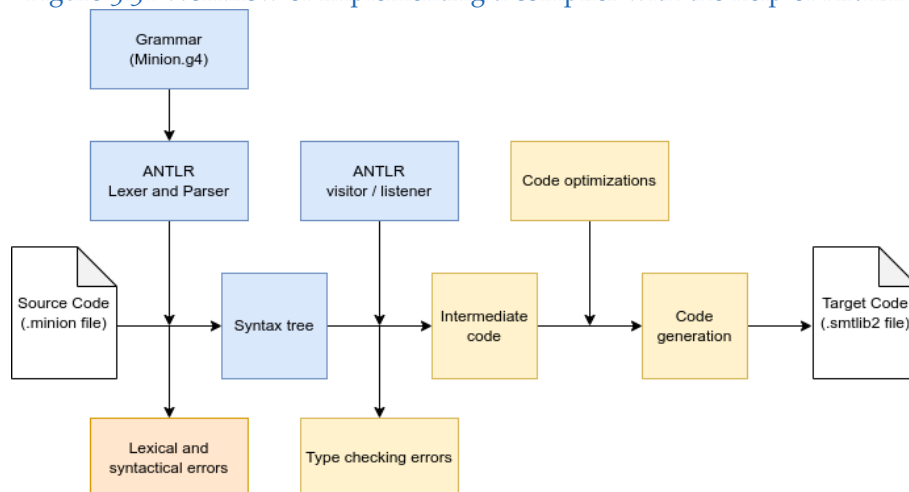
to generate a lexer along with the parser, simplifying the overall parsing process.

[ANTLR](#) is also capable of generating parse trees which can be used to perform various transformations and analyses on the input code. These trees can be transformed into abstract syntax trees (ASTs) that can be used for code generation and optimization.

[ANTLR](#) supports multiple target languages, including Python, Java, C#, and JavaScript, making it a versatile tool for developers working with different programming languages. In addition, it includes various tools for debugging and visualizing the generated parse trees, which can be useful for understanding the parsing process and debugging issues in the generated code. [Koi+19]

When the lexing and parsing are finished, [ANTLR](#) generates visitors and listeners in the desired target programming language. These patterns can be further used to traverse the syntax trees and implement the subsequent steps toward code generation. [Figure 3.3](#) demonstrates the steps necessary to implement a compiler utilizing [ANTLR](#). While the blue marked parts are provided by [ANTLR](#) and generate lexical and syntactical errors (yellow box), the result, which is the syntax tree, can be used for further processing (yellow boxes).

Figure 3.3.: Workflow of implementing a compiler with the help of ANTLR



3. Compiler

The next chapter demonstrates how a compiler can be implemented with the help of [ANTLR](#).

4. Compilation of MINION to SMT-LIB2

This chapter summarizes the practical approach to compiling MINION models into [SMT-LIB2](#) models. Along with a discussion of the compiler architecture, also an overview of the tools and libraries used during development will be listed. In addition, a comprehensive list of the mappings for individual statements will be offered.

4.1. Implementation

The implementation was done using Python3 and [ANTLR](#). Python is a high-level, interpreted programming language. In recent versions, Python has gained significantly in interpretation speed. For this reason and due to its simplicity of syntax, the language has grown extremely popular and versatile. It is used in various applications, such as web development, artificial intelligence, scientific computing, and data analysis. While it is less commonly used for the purpose of compiler construction than some other languages like C++ or Java, it has been used successfully in several compiler construction projects, particularly for creating language-specific tools and utilities. A famous example of Python used for compiler construction is PyPy, a just-in-time Python compiler written in Python [[Tea23](#); [Bol+09](#)]. Another successful approach of compiler development using Python is described in [[JBP19](#)].

As mentioned in [section 3.4](#), the lexer and parser were generated from an input language definition of MINION using the [ANTLR](#) library. Furthermore, annotations were added to the parser rules to simplify walking through the

resulting syntax tree. [Listing A.1](#) lists the full language definition of MINION implemented in [MIN2SMT](#) using the [ANTLR](#) notation.

Together, Python, with its simple syntax and [ANTLR](#) as a parser generator, this combination provides a solid foundation for developing a compiler that translates the MINION input language into [SMT-LIB2](#) format. In addition, Python provides an interface with various testing tools to ensure the quality of the code. [Chapter 5](#) will delve into this topic in more detail.

4.1.1. Used libraries

The following Python3 libraries/packages were used during development:

- *antlr4-python3-runtime* [[VPH23](#)] (4.12.0): This package is the core of the [ANTLR](#) library which empowers developers to utilize the [ANTLR](#)-generated elements like visitors and listeners.
- *antlr4-tools* [[Par23b](#)] (0.1): With these tools, running [ANTLR](#) no longer requires the installation of Java or [ANTLR](#), enabling direct usage with Python. Additionally, this package facilitates the generation of an abstract syntax tree from a specified grammar file and accompanying input file. [Figure 3.1](#) demonstrates an example output of this package.
- *pyinstaller* [[Cor23](#)] (5.7.0): pyinstaller is a package that facilitates the development of platform-independent Python programs. It accomplishes this by bundling the whole Python application as well as all its dependencies into one single executable file. This simplifies the creation of platform-independent release artifacts.
- *pylint* [[Wal23](#)] (2.17.2): pyreverse, as part of the pylint package, was used to generate class diagrams as well as package diagrams. This can be useful for getting and keeping an overview of the overall program structure.

4.2. Architecture of the compiler

The structure of the compiler is divided into several components, which are presented below:

Controller

The Controller class is the entry point of the compiler. It initializes the lexer and parser. After the tree generation, the visit method of [ANTLR](#) is called on this tree. If any type errors occur, the controller handles the output of the error messages. If no errors have occurred, it starts translating the intermediate code and then writes the result, the [SMT-LIB2](#) code, to an output file.

The Controller class can provide a list of all variable declarations for testing purposes. [Section 5.2](#) describes the usage of this list.

MinionCustomVisitor

This visitor class is derived from [ANTLR](#) generated base class. In this class, variable declarations, constraints, and other elements of the MINION language get extracted and stored in symbol tables.

During the creation of the symbol tables as well as after the creation of the intermediate representation of the constraints, type checking is performed. The following errors can be recognized and will be reported to the user:

- double declaration of variables;
- usage of an undefined variable;
- wrong arguments for constraints;
- usage of an undefined or not supported constraint.

The symbol tables are used to create an intermediate code representation, as presented in [listing 4.1](#).

Mappers

There exist two mappers:

- *Constraint Mapper*: creates a new constraint object depending on the constraint's name. Each MINION constraint is mapped to an internal constraint representation.

Listing 4.1: *The farmers problem, represented as MINION intermediate code.*

```

1 Variable("pigs", Type.TYPE_INT, dimensions=[0, 7])
2 Variable("hens", Type.TYPE_INT, dimensions=[0, 7])
3 Constraint_weightedsumgeq(Vector(Const(2), Const(4)),
4                             Vector(Variable("hens"), Variable("pigs")),
5                             Const(22))
6 Constraint_weightedsumleq(Vector(Const(2), Const(4)),
7                             Vector(Variable("hens"), Variable("pigs")),
8                             Const(22))
9 Constraint_sumleq(Vector(Variable("hens"), Variable("pigs")),
10                    Const(7))
11 Constraint_sumgeq(Vector(Variable("hens"), Variable("pigs")),
12                    Const(7))

```

- *Variable Mapper*: builds a symbol table of variables. A mapping is added using the information provided by the variable declaration. Each time a variable is used in the next step, this mapping can be accessed, and the corresponding variable can be queried.

Constraints

The base class `Constraint` takes a name and various arguments as parameters. The class provides different methods for accessing the name and arguments, as well as the `is_valid()` and `to_smtlib2()` methods. Each MINION constraint is represented by a corresponding constraint class which is derived from the class `Constraint` and implements the `is_valid()` and `to_smtlib2()` methods:

- `is_valid()`: This method is used by the `MinionCustomVisitor` class during the type checking. Since each constraint accepts a different number and types of arguments, this information is deposited in this method.
- `to_smtlib2()`: Each constraint holds the information, how it itself has to be translated. This method translates the MINION intermediate representation to [SMT-LIB2](#) intermediate code.

Three special implementations of the Constraint class are the classes WatchedConstraint, SumConstraint and TableConstraint:

- WatchedConstraint: MINION supports two watched constraints: watched-and and watched-or, which ensure that the provided constraints are all true, or at least one of the provided constraints is true, respectively.
- SumConstraint: forms the basis for multiple constraints:
 - sumleq and sumgeq
 - watchsumgeq and watchsumleq
- TableConstraint: As the name suggests, this is the base class for the table and negativetable constraints.

The translation to [SMT-LIB2](#) in each case only differs in the relational and logical operators used for all these constraints. Therefore, their `is_valid()` and `to_smtlib2()` methods are implemented inside these classes.

Arguments

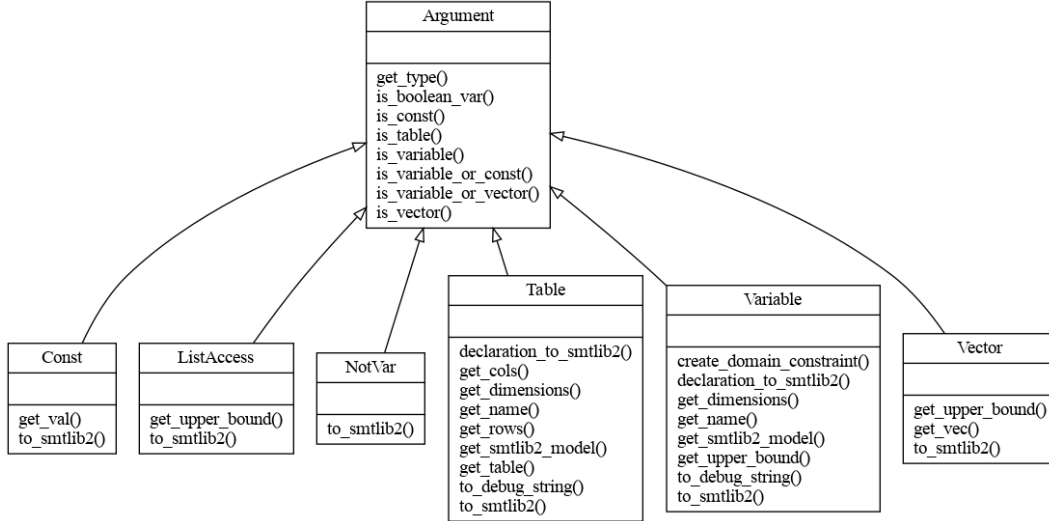
As mentioned before, constraints can have various arguments as parameters. The representation is encoded into an Argument base class which provides some methods for querying the type of the argument. In addition, an internal type is added to the argument object, such as `TYPE_INT` and `TYPE_BOOL`. Each argument is then encoded in a concrete child class, as depicted in [fig. 4.1](#). The following list describes the different argument classes:

- `const`: a constant value, such as -1, 0, 1, 2, ...
- `list_access`: the access of a variable with more than one dimension, e.g., `a[1]`, `A[5]`. The internal type is the type of the accessed variable.
- `not_var`: a representation of the Boolean not operator, e.g. `!a`. A `not_var` has the internal type `TYPE_BOOL`
- `table`: a representation of a table, defined in the tuple list section of a MINION input. It is treated as a variable of internal type `TYPE_INT`.
- `variable`: a simple variable name, e.g. `a`, `var`, `variable`. Variables can have the internal type `TYPE_INT` or `TYPE_BOOL`.

4. Compilation of MINION to SMT-LIB2

- vector: a list of constant values, variables, variable accesses or not_vars, e.g., [1, 2, 3], [a, b], [A[1], 0, !a]. Each element of a vector can have its own internal type.

Figure 4.1.: A class diagram of the arguments class and its derived child classes.



Each argument object gets created by the `MinionCustomVisitor` using its symbol tables. The information about the arguments is also used during type checking.

SMT-LIB2 Statements

As previously mentioned, the `MinionCustomVisitor` generates an intermediate representation of the MINION input code. Additional intermediate steps are necessary to obtain the SMT-LIB2 format. Thus, the MINION intermediate code is transformed into a SMT-LIB2 intermediate representation. Listing 4.2 demonstrates, how the supplementary representation of *The farmers problem* is applied. Furthermore, some optimizations have already been performed on the intermediate code, as outlined in section 5.3.2. This supplementary representation not only aids in the conversion process but also simplifies unit testing by enabling separate testing of each translation step. Another

4. Compilation of MINION to SMT-LIB2

advantage of the two intermediate code levels is the possibility to integrate further target languages later.

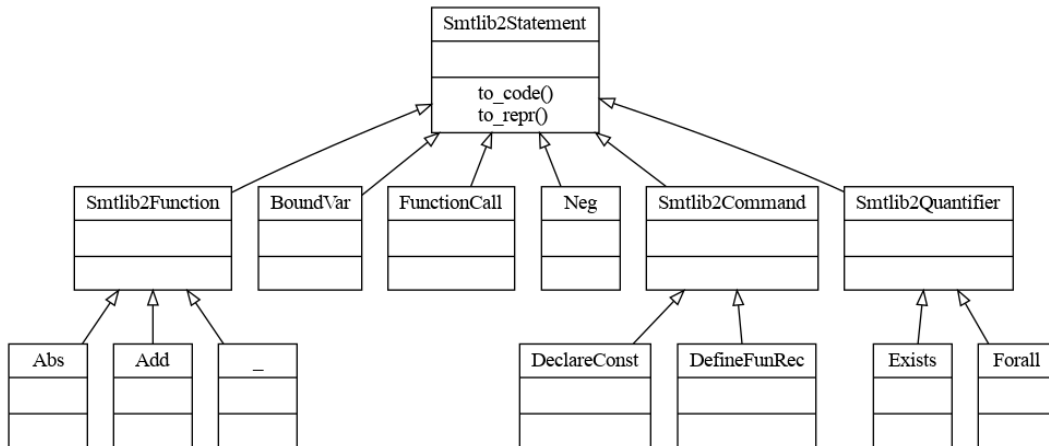
Listing 4.2: *The farmers problem*, represented as SMT-LIB2 intermediate code.

```

1  DeclareConst("pigs", 0, "Int")
2  DeclareConst("hens", 0, "Int")
3  Leq(0, "pigs", 7)
4  Leq(0, "hens", 7)
5  Eq(
6    Add(
7      Mul(2, "hens"),
8      Mul(4, "pigs")
9    ),
10   22
11 )
12 Eq(
13   Add("hens", "pigs"),
14   7
15 )

```

Figure 4.2.: A class diagram of the Smtlib2Statement class and its derived child classes.



As fig. 4.2 demonstrates, multiple inheritance layers were used to build the SMT-LIB2 representation:

- Smtlib2Statement: this base class implements the `to_code()` method

which converts the statement into a string. Furthermore, the statements for *bound variables* (used for quantifiers), *function calls*, and *negations* are derived from the `Smtlib2Statement` class. These classes are not assertable, which means, they cannot be placed inside an `(assert <stmt>)` block separately.

- `Smtlib2Function`: this class implements the `to_code()` method for:
 - arithmetical operations, such as `*`, `/`, `+`, `-`, `pow`, `abs`, `mod`
 - Boolean operations, such as relational or equality operators, `distinct`, `and`, `or`, `not`, `implication`
 - if-then-else constructs
 - select statement, for accessing array elements
- `Smtlib2Command`: this class represents variable declarations and function definitions.
- `Smtlib2Quantifier`: this class represents the `exists` and `forall` quantifiers.

Each `smtlib2_statement` can be converted to the final output string using its `to_code()` method.

4.3. Translation of statements

This section presents the translations from MINION constraints to [SMT-LIB2](#) models. We use the naming conventions listed in [table 4.1](#) to simplify the explanation.

Table 4.1.: Description of the used naming convention.

Symbol	Description
A, B	vector variable
T	table variable
t00, t01, ... t22	table elements
n, m	number of vector elements = A.length - 1 or number of elements of a $n \times m$ vector
v1, v2, ..., vn	vector elements
x, y, z i(ndex) e(lement)	variable or constant values
b	Boolean variable
c	constant values: 1, 2, 3, ...
[c1, c2, ..., cn]	multiple constant values

Although Jefferson et al. [[Jef+14](#)] list all available MINION constraints, this list is incomplete in terms of possible parameters per constraint, and in some cases, the description could be more precise. Therefore, the constraints are listed below again, where the descriptions from [[Jef+14](#)] were adopted and supplemented with additional information with regard to the implementation of [MIN2SMT](#).

Variable declarations

Three types out of four types of MINION variables are supported by [MIN2SMT](#): DISCRETE, SPARSEBOUND, and BOOL. The first two types always have a domain specification. Since Boolean variables inherently have the domain 0..1, no separate domain specification is necessary. Furthermore, variables can also

4. Compilation of MINION to SMT-LIB2

appear as vectors or multidimensional vectors. The listing below demonstrates how each type translates into [SMT-LIB2](#) with different domains and dimensions:

```
1 DISCRETE A[2] {-1..5}
2 DISCRETE a {0..10}
3 SPARSEBOUND sb {1,3,4,5}
4 BOOL b
5 BOOL ab[6]
```

```
1 (declare-const A (Array Int Int))
2 (declare-const a Int)
3 (declare-const sb Int)
4 (declare-const b Bool)
5 (declare-const ab (Array Int Bool))
6
7 ; A[2] {-1..5}
8 (assert (forall ((i Int)) (=> (<= 0 i 1) (<= -1 (select A i) 5))))
9 ; a {0..40}
10 (assert (<= 0 a 40))
11 ; sb {1,3,4,5}
12 (assert (or (= sb 1) (= sb 3) (= sb 4) (= sb 5)))
```

It can be easily seen that for modeling the domain, additional domain constraints are added to the output.

Constraint abs

```
1 abs(x, y)
```

ensures that $x = |y|$. This means that x is the absolute value of y :

```
1 (assert (= x (abs y)))
```

Constraint alldiff

```
1 alldiff(A)
```

ensures that each element of A takes a different value:

```
1 (assert
2   (forall ((i Int) (j Int))
3     (=>
4       (and
5         (< 0 i 5)
6         (< 0 j 5)
7         (= (select A i) (select A j))
8       )
9       (= i j)
10    )
11  )
12 )
```

Constraint alldiffmatrix

```
1 alldiffmatrix(A, c)
```

ensures that in each row of the multidimensional vector A the constant value c appears exactly once:

```
1 (define-fun-rec fun_alldiffmatrix ((i Int) (j Int) (acc Int)) Bool
2   (ite (>= j m)
3     (= acc 1)
4     (ite (= (select A i j) c)
5       (ite (= acc 1)
6         false
7         (fun_alldiffmatrix i (+ j 1) 1)
8       )
9     (fun_alldiffmatrix i (+ j 1) acc)
10  )
```

4. Compilation of MINION to SMT-LIB2

```
11   )
12 )
13 (assert
14   (and
15     (fun_alldiffmatrix 0 0 0)
16     ...
17     (fun_alldiffmatrix n 0 0)
18   )
19 )
```

Constraint difference

```
1 difference(x, y, z)
```

ensures that $z = |x - y|$

```
1 (assert (= (abs (- x y)) z))
```

Constraint diseq

```
1 diseq(x, y)
```

ensures that the two variables have a different value:

```
1 (assert (distinct x y))
```

Constraint div

```
1 div(x, y, z)
```

ensures that $\lfloor \frac{x}{y} \rfloor = z$ and is always false in case of $y = 0$:

```

1 (assert
2   (and
3     (distinct y 0)
4     (ite
5       (or
6         (and (> x 0) (> y 0))
7         (and (< x 0) (> y 0))
8       )
9       (= z (div x y))
10      (=
11        (div (- x) (- y))
12        z
13      )
14    )
15  )
16 )

```

The MINION implementation of the division differs from the standard implementation of the division in [SMT-LIB2](#). Therefore, a special treatment had to be implemented. Jefferson et al. [Jef+14] brings the following example:

$$10/3 = 3 \tag{4.1}$$

$$(-10)/3 = -4 \tag{4.2}$$

$$10/-3 = -4 \tag{4.3}$$

$$(-10)/-3 = 3 \tag{4.4}$$

Constraint element

```

1 element(A, i, e)

```

ensures that $A[i] = e$, where $0 \leq i \leq |A|$. The constraint is *false*, if i is outside the index range and $e = 0$:

```

1 (assert
2   (and

```

4. Compilation of MINION to SMT-LIB2

```
3      (= (select A i) e)
4      (< i n)
5    )
6  )
```

Constraint `element_one`

```
1 element_one(A, i, e)
```

is identical to [Constraint `element`](#). However, A is indexed from 1:

```
1 (assert
2   (and
3     (= (select A (- i 1)) e)
4     (< (- i 1) n)
5   )
6 )
```

Constraint `eq`

```
1 eq(x, y)
```

ensures that two variables take equal values:

```
1 (assert (= x y))
```

Since MINION Boolean values are represented with 0/1 values and, on the contrary, [SMT-LIB2](#) expects *true* and *false* for boolean constraints, the values have to be translated properly:

```
1 BOOL b1
2 BOOL b2
3 eq(b1, 0)
4 eq(1, b2)
```

therefore translates to

```
1 (assert (= b1 false))
2 (assert (= b2 true))
```

Constraint gacalldiff

See [Constraint alldiff](#).

Constraint hamming

```
1 hamming(A, B, c)
```

ensures that the hamming distance between A and B is greater or equal to c . This means $\sum_i A[i] \neq B[i] \geq c$

```
1 (assert
2   (>=
3     (+
4       (ite (distinct (select A 0) (select B 0)) 1 0)
5       ...
6       (ite (distinct (select A n) (select B n)) 1 0)
7     )
8     c
9   )
10 )
```

Constraint ineq

```
1 ineq(x, y, c)
```

ensures that $x \leq y + c$. This constraint can be used to express $x < y$ iff $c = -1$:

```
1 (assert (<= x (+ y c)))
```

Constraint lexleq

```
1 lexleq(A, B)
```

ensures that A is lexicographically less than or equal to B , where A and B are both of same length:

```
1 (define-fun-rec fun_lexleq ((i Int)) Bool
2   (ite
3     (>= i n)
4     true
5     (ite
6       (> (select A i) (select B i))
7       false
8       (ite
9         (< (select A i) (select B i))
10        true
11        (fun_lexleq (+ i 1))
12      )
13    )
14  )
15 )
16 (assert (fun_lexleq 0))
```

Constraint lexless

```
1 lexless(A, B)
```

ensures that A is lexicographically less than B , where A and B are both of same length:

4. Compilation of MINION to SMT-LIB2

```
1 (define-fun-rec fun_lexless ((i Int)) Bool
2   (ite
3     (>= i n)
4     false
5     (ite
6       (> (select A i) (select B i))
7       false
8       (ite
9         (< (select A i) (select B i))
10        true
11        (fun_lexless (+ i 1))
12      )
13    )
14  )
15 )
16 (assert (fun_lexless 0))
```

Constraint litsumgeq

```
1 litsumgeq(A, [1,2,3,4,5], 3)
```

ensures that there exists at least c distinct indices i such that $A[i] = B[i]$.
This means $\sum_i A[i] = B[i] \geq c$:

```
1 (assert
2   (>=
3     (+
4       (ite (= (select A 0) v1) 1 0)
5       ...
6       (ite (= (select A n) vn) 1 0)
7     )
8     c
9   )
10 )
```


Constraint max

```
1 max(A, x)
```

ensures that x is equal to the maximum of any element of A :

```
1 (assert
2   (and
3     (or
4       (= x (select A 0))
5       ...
6       (= x (select A n))
7     )
8     (>= x (select A 0))
9     ...
10    (>= x (select A n))
11  )
12 )
```

Constraint min

```
1 min(A, x)
```

ensures that x is equal to the minimum of any element of A :

```
1 (assert
2   (and
3     (or
4       (= x (select A 0))
5       ...
6       (= x (select A n))
7     )
8     (<= x (select A 0))
9     ...
10    (<= x (select A n))
11  )
```

4. Compilation of MINION to SMT-LIB2

12)

Constraint minuseq

1 minuseq(x, y)

ensures that $x = -y$:

1 (assert (= x (- y)))

Constraint modulo

1 modulo(x, y, z)

ensures that $x \bmod y = z$. The constraint is always *false* when $y = 0$.

```
1 (assert
2   (and
3     (distinzt y 0)
4     (ite
5       (or
6         (and (> x 0) (> y 0))
7         (and (< x 0) (> y 0))
8       )
9       (= (mod x y) z)
10      (ite
11        (and (> x 0) (< y 0))
12        (=
13          (mod (- x) (- y))
14          (- z)
15        )
16        (=
17          (mod (- x) y)
18          (- z)
19        )
20      )
21    )
22  )
```

4. Compilation of MINION to SMT-LIB2

```
19      )
20    )
21  )
22 )
23 )
```

The MINION implementation of the modulo operator differs from the standard implementation of the modulo operator in [SMT-LIB2](#). Therefore, a special treatment had to be implemented. Jefferson et al. [[Jef+14](#)] brings the following example:

$$3\%5 = 3 \quad (4.5)$$

$$-3\%5 = 2 \quad (4.6)$$

$$3\% - 5 = -2 \quad (4.7)$$

$$-3\% - 5 = -3 \quad (4.8)$$

Constraint `mod_undefzero`

is identical to [Constraint modulo](#), except the constraint is always *true* when $y = 0$:

Constraint `negativetable`

```
1 negativetable(A, T)
```

ensures that there exists no column t in table T , such that $t = A$. Tables are defined in the `**TUPLELIST**` section:

```
1 (assert
2   (and
3     (or
4       (distinct t00 (select A 0))
5       (distinct t01 (select A 1))
6       (distinct t02 (select A 2))
7     )
  )
```

4. Compilation of MINION to SMT-LIB2

```
8      (or
9        (distinct t10 (select A 0))
10       (distinct t11 (select A 1))
11       (distinct t12 (select A 2))
12     )
13     (or
14       (distinct t20 (select A 0))
15       (distinct t21 (select A 1))
16       (distinct t22 (select A 2))
17     )
18   )
19 )
```

Constraint occurrence

```
1 occurrence(A, c, x)
```

ensures that the value x occurs exactly c times in A :

```
1 (assert
2   (=
3     c
4     (+
5       (ite (= (select A 0) x) 1 0)
6       ...
7       (ite (= (select A n) x) 1 0)
8     )
9   )
10 )
```

Constraint occurrencegeq

```
1 occurrencegeq(A, c, x)
```

ensures that the value x occurs at least c times in A :

```
1 (assert
2   (>=
3     c
4     (+
5       (ite (= (select A 0) x) 1 0)
6       ...
7       (ite (= (select A n) x) 1 0)
8     )
9   )
10 )
```

Constraint `occurrenceleq`

```
1 occurrenceleq(A, c, a)
```

ensures that the value x occurs at most c times in A :

```
1 (assert
2   (<=
3     c
4     (+
5       (ite (= (select A 0) x) 1 0)
6       ...
7       (ite (= (select A n) x) 1 0)
8     )
9   )
10 )
```

Constraint `pow`

```
1 pow(x, y, z)
```

ensures that $x^y = z$:

```
1 (assert
2   (ite
3     (= y 0)
4     (= z 1)
5     (= z (^ x y))
6   )
7 )
```

Constraint product

```
1 product(x, y, z)
```

ensures that $x \cdot y = z$

```
1 (assert (= z (* x y)))
```

For Boolean variables, the product constraint acts as conjunction:

```
1 (assert (= z (and x y)))
```

Constraint reify

```
1 reify(Constraint, b)
```

ensures that the constraint must be satisfied if $b = true$ and the constraint must not be satisfied if $b = false$:

```
1 (assert (= b Constraint))
```

Constraint reifyimply

```
1 reifyimply(Constraint, b)
```

ensures, an implication between b and the constraint which means
 $b \Rightarrow \text{Constraint}$:

```
1 (assert (=> b Constraint))
```

Constraint sumgeq

```
1 sumgeq(A, x)
```

ensures that $\sum_i A_i \geq c$:

```
1 (assert
2   (>=
3     (+
4       (select A 0)
5       ...
6       (select A n)
7     )
8     x
9   )
10 )
```

Constraint sumleq

```
1 sumleq(A, x)
```

ensures that $\sum_i A_i \leq c$:

4. Compilation of MINION to SMT-LIB2

```
1 (assert
2   (<=
3     (+
4       (select A 0)
5       ...
6       (select A n)
7     )
8     x
9   )
10 )
```

Note that MINION does not have a `sumeq` constraint. [Section 5.3.2](#) solves this problem.

Constraint table

```
1 table(A, T)
```

ensures that there exists at least one column t in table T , such that $t = A$. Tables are defined in the `**TUPLELIST**` section.

```
1 (assert
2   (or
3     (and
4       (= t00 (select A 0))
5       (= t01 (select A 1))
6       (= t02 (select A 2))
7     )
8     (and
9       (= t10 (select A 0))
10      (= t11 (select A 1))
11      (= t12 (select A 2))
12     )
13     (and
14       (= t20 (select A 0))
15       (= t21 (select A 1))
16     )
17   )
18 )
```


4. Compilation of MINION to SMT-LIB2

```
16      (= t22 (select A 2))
17    )
18  )
19 )
```

Constraint `w_inintervalset`

```
1 w-inintervalset(a, [c1, c2, c4, c5])
```

ensures that $c1 \leq x \leq c2, c3 \leq x \leq c4 \dots$ holds. The interval list must be given in numerical (strictly monotonously rising) order:

```
1 (assert (or (<= c1 a c2) (<= c4 a c5)))
```

Constraint `w_inrange`

```
1 w-inrange(x, [c1, c2])
```

ensures that $c1 \leq x \leq c2$:

```
1 (assert (<= c1 x c2))
```

Constraint `w_inset`

```
1 w-inset(x, A)
```

ensures that x equals one of the values in the given set:

```
1 (assert
2   (or
3     (= x v1)
4     ...
```

4. Compilation of MINION to SMT-LIB2

```
5      (= x vn)
6    )
7  )
```

Constraint `w_literal`

```
1 w_literal(x, c)
```

ensures that $x = c$:

```
1 (assert (= x c))
```

Constraint `w_notinrange`

```
1 w_notinrange(x, [c1, c2])
```

ensures that $x < c1$ or $x > c2$:

```
1 (assert (or (< x c1) (> x c2)))
```

Constraint `w_notinset`

```
1 w_notinset(x, [c1, c2, ..., cn])
```

ensures that x is not equal to any of the values in the given set:

```
1 (assert
2   (and
3     (distinct x c1)
4     (distinct x c2)
5     ...
6     (distinct x cn))
```

4. Compilation of MINION to SMT-LIB2

```
7   )  
8 )
```

Constraint `w_notliteral`

```
1 w-notliteral(x, c)
```

ensures that $x \neq c$:

```
1 (assert (distinct x c))
```

Constraint `watched-and`

```
1 watched-and({Constraint1, Constraint2, ..., Constraintn})
```

ensures that all constraints are *true*. This constraint may be used in combination with [Constraint `reify`](#).

```
1 (assert  
2   (and  
3     Constraint1  
4     Constraint2  
5     ...  
6     Constraintn  
7   )  
8 )
```

Constraint `watched-or`

```
1 watched-or({Constraint1, Constraint2, ..., Constraintn})
```

ensures that at least one of the constraints is *true*:

```
1 (assert
2   (or
3     Constraint1
4     Constraint2
5     ...
6     Constraintn
7   )
8 )
```

Constraint `watchelement`

See [section 4.3](#).

Constraint `watchelement_one`

See [section 4.3](#).

Constraint `watchelement_undefzero`

```
1 watchelement_undefzero(A, i, e)
```

ensures that $A[i] = e$, where $0 \leq i \leq |A|$. The constraint is *true*, if i is outside the index range and $e = 0$:

```
1 (assert
2   (or
3     (= (select A i) e)
4     (and
5       (>= i n)
6       (= e 0)
7     )
8   )
9 )
```

Constraint watchless

```
1 watchless(x, y)
```

ensures that $x < y$:

```
1 (assert (< x y))
```

Constraint watchsumgeq

```
1 watchsumgeq(A, c)
```

See [Constraint sumgeq](#)

Constraint watchsumleq

```
1 watchsumleq(A, c)
```

See [Constraint sumleq](#)

Constraint watchvecneq

```
1 watchvecneq(A, B)
```

ensures that A and B are not the same. This means $\exists i : A[i] \neq B[i]$:

```
1 (assert
2   (or
3     (distinct (select A 0) (select B 0))
4     ...
5     (distinct (select A n) (select B n))
6   )
7 )
```

Constraint `weightedsumgeq`

```
1 weightedsumgeq([c0, c1, ..., cn], A, x)
```

ensures that $\sum_i A_i \cdot c_i \geq x$:

```
1 (assert
2   (>=
3     (+
4       (* c1 (select A 0))
5       (* c2 (select A 1))
6       ...
7       (* cn (select A n)
8     )
9     x
10  )
11 )
```

Constraint `weightedsumleq`

```
1 weightedsumleq([c0, c1, ..., cn], A, x)
```

ensures that $\sum_i A_i \cdot c_i \leq x$:

```
1 (assert
2   (<=
3     (+
4       (* c1 (select A 0))
5       (* c2 (select A 1))
6       ...
7       (* cn (select A n)
8     )
9     x
10  )
11 )
```

Note that MINION does not have a `weightedsumeq` constraint. [Section 5.3.2](#) solves this problem.

4.4. Limitations

Currently, the MINION constraints listed below are not supported because they implement unique algorithms or address CSP-specific issues, for example, GAC. GAC stands for *Generalized Arc Consistency*, a constraint propagation technique used in constraint satisfaction problems. [SMT-LIB2](#) does not have a built-in mechanism for expressing GAC, as it is primarily focused on first-order logic and [SMT](#) theories. However, some [SMT](#) solvers may implement GAC internally to improve their performance on specific types of constraints.

- `gacschema`
- `gcc`
- `gccweak`
- `haggisgac`
- `haggisgac_stable`
- `lighttable`
- `mddc`
- `negativemddc`
- `shortstr2`
- `str2plus`

Furthermore, some MINION constraints that explicitly enforce GAC but are otherwise identical to other constraints were not implemented separately but only mapped to their companion constraints. This applies to the `watchelement`, `watchelement_one`, and `gacalldiff` constraints.

[Min2SMT](#) does currently not support the `**SEARCH**` and `**SHORTTUPLELIST**` sections. The search section, on the one hand, allows defining a variable ordering for the output, defining a value ordering and an objective function, and specifying which variables should be printed. However, [SMT-LIB2](#) does not provide any of these features. [Dav13] On the other hand, the short

tuple list section is not used since no constraints that accept short tuple lists were implemented.

5. Testing, optimizations, and performance

Software testing is a crucial part of software development. A good testing framework simplifies software maintenance and the integration of new software modules and units. Furthermore, it can be used for detecting deviations from the software specification as well as for reducing the probability of occurrence of errors in production.

This chapter gives an overview of the testing techniques alongside the test data used during the development of [MIN2SMT](#). Furthermore, the performed optimizations will be discussed. The end of the chapter summarizes the performance of the resulting [SMT-LIB2](#) files in combination with performed optimizations.

5.1. Unit testing

Unit testing is a software development practice that involves testing individual units or components of an application in isolation, independent of the rest of the system. The primary objective of unit testing is to confirm that each unit of code functions as intended and that any errors or defects are detected and fixed early in the development process.

Test Driven Development (TDD), or Test First Development, is a commonly used approach to integrate unit testing into the development process and ensure maximum code coverage. This well-established software development process is frequently utilized in agile development and was employed during the creation of [MIN2SMT](#).

[Bec02] describes the TDD workflow as follows:

1. *Add a little test*: The technique's name implies that the test is written before implementing a new feature.
2. *Run all tests and fail*: Since the feature is not implemented yet, the new test will fail.
3. *Make a little change*: In this step, the feature is implemented.
4. *Run the tests and succeed*: After the implementation, all tests must pass.
5. *Refactor to remove duplication*: Once all tests are positive, the recently implemented features can be refactored and duplicates removed. Even a major architecture change should not break the tests. Otherwise, the procedure has to be restarted from step 2.

TDD was utilized to conduct unit testing on all classes and components involved in the compilation process, including the intermediate code representation. The unique characteristic of unit tests is that they serve as software specifications stored as tests. In the case of refactoring, these tests provide an ideal foundation for verifying whether the specifications are still being met despite significant code modifications.

5.2. Integration testing

Although unit tests assess the most basic functionalities, it is also essential to devise a method to test the entire translation process as a larger unit. The main aim is to check whether both solvers (MINION and Z₃) produce the same results and, thus, whether the MINION input and the compiled SMT-LIB2 output are equivalent. In order to achieve this observation, two cases have to be distinguished:

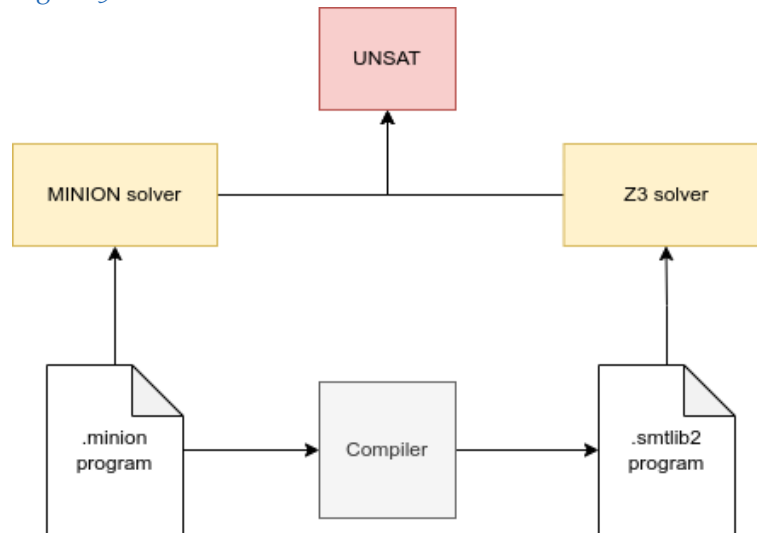
1. MINION is not able to find any solution. In this case, the Z₃ solver must also produce the result UNSAT.
2. If the MINION solver yields one or more solutions, then the Z₃ solver must also yield SAT when the variables are asserted with the provided models.

For verification of this equivalency, an integration test framework was developed. Figure 5.2 and fig. 5.1 demonstrate the workflow of the integration test framework for both cases:

Case UNSAT

1. The MINION code is run by the MINION solver. No solution can be found; the result is **UNSAT**.
2. The MINION code gets translated by the **MIN2SMT** compiler.
3. The resulting **SMT-LIB2** file is run by the **Z3** solver. The result must also be **UNSAT**.

Figure 5.1.: Workflow of the test framework in the UNSAT case.



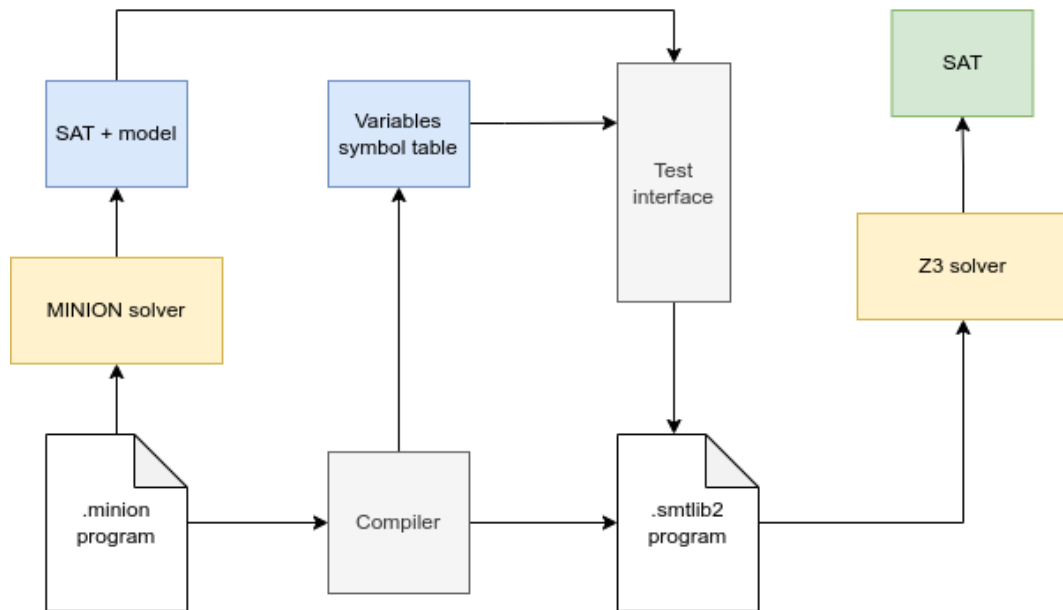
Case SAT

1. The MINION code is run by the MINION solver, using the `-findallsols -sollimit 5` flags. If this flags are set, MINION tries to find as much solutions as possible, but at most 5.
2. The integration test interface parses the MINION output.

3. The MINION code gets translated by the [MIN2SMT](#) compiler. As a by-product, the compiler passes the symbol table of variables to the test interface.
4. After merging the solutions from step 2 with the variable metainformation from step 3, the test interface injects the resulting data into the [SMT-LIB2](#) code.
5. The resulting [SMT-LIB2](#) file is run by the [Z3](#) solver. The result must also be [SAT](#).
6. The process outlined in steps 4 and 5 is repeated for each resulting solution.

For the test framework to provide a correct output, the entire `**SEARCH**` section must be removed from the MINION code. Otherwise, automatically mapping the variables with the corresponding values would not work, as this section defines which variables have to be output and in which order.

Figure 5.2.: Workflow of the test framework in the SAT case.



5.2.1. Test data

As already mentioned, the test framework uses MINION programs as input. Therefore, we gathered a large collection of different categories of programs to enable thorough integration testing. The test data is composed of the following categories, denoted with TS_n :

- **TS₁** Constraint testcases: For each supported MINION constraint we have written one or multiple test files. Each file uses different types of parameters for one specific constraint, such as variables, constant values, vectors, and tables. Additionally, different variable types (DISCRETE and BOOL) were tested. For example, the constraint `eq` can be used with the following parameters:

```

1      eq(1, 2)
2      eq(a, 1)
3      eq(1, a)
4      eq(A[0], a)
```

where `a` is an arbitrary variable of type DISCRETE or BOOL and `A` is a variable with multiple dimensions. Of course, some more permutations of provided arguments are possible.

- **TS₂** MINION in Practice: These are MINION files taken from [Jef+14] and include various complex CSPs or logic puzzles such as *The farmers problem* described in section 2.1.1, *The Zebra Puzzle* or the *N-Queens problem* which has been discussed in detail in [Höf15].
- **TS₃** ISCAS85 data set: These input files were used in [Hof14]. They mainly utilize the following constraints: `diseq`, `reify`, `reifyimply`, `eq`, `max`, `min`, `sumgeq` and `sumleq`. In addition, a larger set of variables (between 300-4500) and one large vector (up to 2300 elements) are used.
- **TS₄** Mutation testing examples: Wotawa, Nica, and Aichernig utilized these files in [WNA10] for experimental and benchmarking activities. The following constraints were used in these files: `watched-or`, `eq`, `ineq`, `reify`, `element`, `watchsumgeq`, `watchsumleq`, `sumgeq`, `sumleq`.
- **TS₅** Spreadsheet evaluation data: These MINION files were generated automatically from spreadsheets. Hofer et al. [Hof+] utilized the files

for fault localization in spreadsheets. The evaluation data of their experiments are located at the corresponding Git repository [[Hof23](#)].

Overall, more than 1700 test files were used for extensive integration testing.

5.3. Optimizations

Optimization is vital in compiler construction, enhancing the generated code's performance. The main goal of optimization is to minimize the number of instructions executed by the target machine while ensuring that the program's behavior remains unchanged. As a consequence, the performance of the optimized compiled code should be better by means of memory consumption or execution time. Furthermore, as Hofer [[Hof18](#)] pointed out, further optimization goals can be power consumption reduction, network traffic reduction, or database traffic reduction. However, the challenges of compiler optimization are that the correctness must still be granted, and the compilation should be finished after a reasonable compilation time.

5.3.1. Removal of unused variables

Unused variables unnecessarily lengthen the translated code. Additionally, these variables would also be included in the solution finding. This can affect the duration of the solution-finding process and generate more solutions than necessary. Therefore, the symbol table of variables was used to check which variables are used in the further program flow. If variables are not used, they are filtered out automatically. For [MIN2SMT](#) to use this function, the command line parameter *optimize* has to be used.

5.3.2. Geq/Leq Optimization

The following equations can be represented as MINION constraints as shown in [listing 5.1](#).

$$(a \times 4) + (b \times 2) \leq 22 \quad (5.1)$$

$$(a \times 4) + (b \times 2) \geq 22 \quad (5.2)$$

[Listing 5.1](#): Simple equations, rewritten as MINION constraints

1	<code>weightedsumgeq([2, 4], [a, b], 22)</code>
2	<code>weightedsumleq([2, 4], [a, b], 22)</code>

As [\[Jef+14\]](#) states, MINION does not support a weighted sum constraint. However, an optimization step was implemented in [MIN2SMT](#) to replace `sumgeq` / `sumleq`, `watchsumgeq` / `watchsumleq` and `weightedsumgeq` / `weightedsumleq` pairs with corresponding `...sumeq` constraints. The prerequisite for this optimization step is that related `geq/leq` constraints occur in the input code using the exact same arguments. This reduces the code size of the [SMT-LIB2](#) code since only one remaining constraint has to be translated after the optimization.

5.3.3. Sum Constraints

The sum constraints `sumgeq`, `sumleq`, `watchsumgeq`, `watchsumleq`, but also the constraints `occurrence`, `occurrencegeq`, `occurrenceleq` and `hamming` were implemented to use the same translation of the summation logic. The original implementation was optimized using multiple iterations, not only in terms of code length but also in terms of execution time.

The following sections describe the iterations of the solution approaches. A detailed discussion about the optimization results follows in [section 5.4](#).

Solution 1: Initial implementation of the sum function

The initial translation is presented in [listing 5.2](#). However, the execution of many tests, especially from test set TS₃, had to be canceled manually, as Z₃ sometimes took several hours to find a solution.

Listing 5.2: The initial implementation of the sum function.

```
1 (declare-fun sum (Int) Int)
2 (assert
3   (= (sum -1) 0)
4 )
5 (assert
6   (and
7     (>= (sum 4) 5)
8     (forall ((i Int))
9       (=> (<= 0 i 4)
10        (=
11          (sum i)
12          (+
13            (sum (- i 1))
14            (select A i))
15          )
16        )
17     )
18 )
19 )
```

Solution 2: Recursive sum function

As the initial solution was unsatisfactory, another solution had to be found. Z₃ supports the `define-fun-rec` statement which allows to define recursive functions. Therefore, the initial solution was rewritten into a recursive function; see [listing 5.3](#).

Listing 5.3: A recursive variant of the sum function.

```
1 (define-fun-rec sum ((i Int)) Int
2   (ite (< i 0)
3     0
4     (+
5       (select A i)
6       (sum (- i 1))
7     )
8   )
9 )
10 (assert (>= (sum 4) 5))
```

Solution 3: Tail recursive sum function

As Hofer [Hof18] explained, the so-called *Tail-call optimization* can significantly reduce the execution time compared to a regular recursive solution. The call happens just before a function's return, e.g., `return g(x, y)`. The idea of tail-call optimization is to reuse the calling stack frame and therefore avoid overwriting the return address. This saves stack space and decreases execution time. The performance can be compared with a regular while loop in structured programming languages. Therefore, solution 2 was rewritten using the tail recursive approach; see [listing 5.4](#).

Listing 5.4: A tail recursive sum function

```
1 (define-fun-rec sum ((i Int) (acc Int)) Int
2   (ite (< i 0)
3     acc
4     (sum
5       (- i 1)
6       (+ acc (select A i))
7     )
8   )
9 )
10 (assert (>= (sum 4 0) 5))
```

Solution 4: Loop unrolling of the sum function

Aho et al. [Aho+06] explained that loop unrolling could be a useful method to overcome the scheduling limitations of regular loops since global scheduling algorithms are enabled to find more parallelism. With the disadvantage that more instructions than in the original code are necessary, the execution time can be significantly reduced. Listing 5.5 implements this loop unrolling technique. As a consequence, this optimization dramatically speeds up the execution of the tests.

Listing 5.5: The sum function, rewritten using loop unrolling.

```
1 (assert
2   (>=
3     (+
4       (select A 0)
5       (select A 1)
6       (select A 2)
7       (select A 3)
8       (select A 4)
9     )
10    5
11  )
12 )
```

5.3.4. Table Constraints

The table constraints `table` and `negativetable` are special constraints since they do not refer to the `**VARIABLES**` section, but the tables are defined in the `**TUPLELIST**` section. Listing 5.6 gives an example of the usage of the table constraint, where a 2×2 table with the identifier `A` is used.

Solution 1: Initial implementation of the table constraint

The initial implementation was done using `SMT-LIB2`'s `exists` quantifier. It says that at least one row in the given table must exist that equals the vector

Listing 5.6: A sample MINION program that uses the table constraint.

```
1 MINION 3
2 **VARIABLES**
3 BOOL a
4
5 **TUPLELIST**
6 A 2 2
7 1 2
8 3 4
9
10 **CONSTRAINTS**
11 table([a, 2], A)
12
13 **EOF**
```

temp. To accomplish this, the table must be treated like a vector, where an internal name is used as an identifier. In addition, each value must be initially assigned to each cell of the multidimensional array. Listing 5.7 shows the full translation of the table constraint using the described approach. However, there were cases where this solution did not provide the desired output. In addition, the performance of this solution was also not satisfactory. Therefore, an alternative solution had to be found.

Listing 5.7: Initial implementation of the table constraint.

```
1 (declare-const temp (Array Int Int))
2 (assert (= (select temp 0) 1))
3 (assert (= (select temp 1) 2))
4 (assert
5   (exists ((i Int))
6     (and (< 0 i 2)
7          (= (select table i) temp)
8        )
9   )
10 )
```

Solution 2: Loop unrolling

To improve performance, a partial loop unrolling was applied. In this approach, each table column is compared with the vector `temp`, and the results are linked with the `or` operator. It is not hard to guess that, similar to what is described in [section 5.3.3](#), the generated code grows significantly longer. However, this approach found the correct solutions in any case, but a performance improvement was not yet noticeable. This approach is demonstrated in [listing 5.8](#).

Listing 5.8: Translation of the table constraint using partial loop unrolling.

```
1 (declare-const temp (Array Int Int))
2 (assert (= (select temp 0) 1))
3 (assert (= (select temp 1) 2))
4 (assert (or (= (select A 0) temp) (= (select A 1) temp)))
```

Solution 3: Elimination of array read-accesses

Tables are defined with a fixed number of rows and columns and concrete values, making optimization tasks easier. The pre-initialization allows the compiler itself to already prepare the individual values accordingly. This fact was used for the most performant approach: Whole columns are no longer compared with each other; however, each individual values. This results in a disjunction of conjunctions of value comparisons, as shown in [listing 5.9](#). Furthermore, the read operator for [SMT-LIB2](#) arrays `select` was eliminated since each read access of an array is time-consuming. As a consequence, tables no longer have to be stored as an array using a temporary variable; however, the values of the table are inserted directly in the assertion at compile time. Although the code length of the generated code obviously continues to increase, this elimination was finally the decisive optimization factor that significantly increased the performance of the generated code.

Listing 5.9: Translation of the table constraint with eliminated array read-accesses.

```
1 (assert
2   (or
3     (and
4       (= 1 (ite a 1 0))
5       (= 2 2)
6     )
7     (and
8       (= 3 (ite a 1 0))
9       (= 4 2)
10    )
11  )
12 )
```

5.4. Performance discussion

In this section, the performance of the optimizations will be discussed based on the results of the integration tests introduced in [section 5.2](#). The integration tests were executed using a computer with an AMD®Ryzen 5 5625u processor (4.3 GHz, six cores) with 16 GB RAM. The operating system was Ubuntu 22.04, 64-bit.

5.4.1. Geq/Leq Optimization

The Geq/Leq optimization described in [section 5.3.2](#) reduces the code size by the translation of one constraint. However, the disadvantage of this optimization method is that the compile time is significantly increased. For the test cases listed in [table 5.1](#), for example, the compilation process for the optimized variant takes almost 8 seconds, while it only takes a little more than 4 seconds for the non-optimized variant. Furthermore, this table demonstrates that for test cases with the result [SAT](#), there is hardly any difference in whether the optimization is active. Only in those test cases where the result is [UNSAT](#) and the search takes a long time is the search duration significantly reduced in most cases. This optimization method

shows well how compile time, execution speed, and code length can be related to each other as optimization parameters.

Table 5.1.: A selection of test cases from test set **TS₃** with a comparison of the performance with and without Geq/Leq optimization.

Testcase	optimized	not optimized	result
test_c6288_tc_1_81	1.33s	1.23s	SAT
test_c6288_tc_1_82	1.25s	1.30s	SAT
test_c6288_tc_1_83	1.14s	1.30s	SAT
test_c6288_tc_1_84	1.15s	1.21s	SAT
test_c6288_tc_1_85	1.14s	1.16s	SAT
test_c6288_tc_1_86	1.14s	1.23s	SAT
test_c6288_tc_1_87	1.26s	1.17s	SAT
test_c6288_tc_1_88	1.26s	1.17s	SAT
test_c6288_tc_1_89	1.16s	1.17s	SAT
test_c6288_tc_1_90	1.15s	1.28s	SAT
test_c6288_tc_2_81	15m 35s	26m 6s	UNSAT
test_c6288_tc_2_82	17m 1s	30m 57s	UNSAT
test_c6288_tc_2_83	13m 15s	15m 27s	UNSAT
test_c6288_tc_2_85	24m 51s	20m 51s	UNSAT
test_c6288_tc_2_86	4m 11s	5m 16s	UNSAT
test_c6288_tc_2_87	17m 58s	15m 9s	UNSAT
test_c6288_tc_2_88	21m 56s	21m 15s	UNSAT
test_c6288_tc_2_89	1m 47s	1m 33s	UNSAT
test_c6288_tc_2_90	2m 40s	1m 56s	UNSAT
test_c6288_tc_3_81	3m 16s	28m 25s	UNSAT

5.4.2. Sum Constraints

As already described, multiple optimization iterations were necessary to gain an acceptable result for the implementation of the sum constraints. [Table 5.2](#) and [table 5.3](#) present a performance comparison of the four solutions. Different randomly chosen test cases from the **TS₃** test data were used for benchmarking. It can be seen that solutions 1-3 have brought hardly any

performance improvements. Only *solution 4* achieved a significant performance boost, especially in the case of **UNSAT** as a result. In the case of **SAT**, the execution time remains almost constant over all four solutions.

As [table 5.4](#) demonstrates, the execution can still take a long time, especially when there are many variables and an array with a large size in the input file. If the result is **UNSAT**, searching for a solution can still take a long time despite optimizations.

5.4.3. Table Constraints

For the execution of the integration test cases that test the table constraint, the maximal execution time of the MINION solver was limited to 90 seconds, and for **Z3**, it was limited to 3 minutes. This means that the corresponding test failed if no equivalent solution could be found within this time range.

[Table 5.5](#) presents the execution times of test cases from the test set **TS5** for the initial solution and the optimization presented previously. As the table shows, the optimization has led to a significant improvement in performance in all cases. Tests that initially timed out and all other tests (**SAT** and **UNSAT**) are usually executed in less than 1 second after the optimization.

5.4.4. Summary

In summary, the optimizations shown deliver different improvements. On the one hand, in the execution speed, and on the other hand, in the code length. Especially with larger test files, one can easily see the effect of the optimizations. However, there are also some situations in which optimizations have hardly any effect or even lead to side effects, such as a much higher compile time.

5. Testing, optimizations, and performance

Table 5.2.: A selection of test cases from test set **TS₃** with a performance comparison after each optimization step. In addition, the array size and the number of variables are listed.

Testcase	solution 1	solution 2	solution 3	solution 4	result	# array	# var.
c880_tc_1_27	726ms	723ms	718ms	679s	SAT	383	826
c880_tc_1_23	751ms	790ms	786ms	709ms	SAT	383	826
c3540_tc_1_62	5.12s	4.93s	4.76s	4.94s	SAT	1669	3388
c5315_tc_1_78	9.24s	8.77s	9.11s	9.19s	SAT	2307	4792
c499_tc_2_17	1.59s	1.66s	1.66s	347ms	UNSAT	202	445
c499_tc_2_15	1.52s	1.49s	1.49s	428ms	UNSAT	202	445
c880_tc_3_30	8.10s	7.99s	8.01s	886ms	UNSAT	383	826
c880_tc_3_25	10.47s	10.55s	10.4s	824ms	UNSAT	383	826
c880_tc_3_24	13.36s	13.49s	13.36s	1.16s	UNSAT	383	826
c880_tc_2_29	12.83s	12.79s	13.02s	857ms	UNSAT	383	826
c880_tc_2_27	17.24s	17.22s	17.16s	1.20s	UNSAT	383	826
c880_tc_2_24	12.89s	12.85s	12.85s	808ms	UNSAT	383	826
c880_tc_2_22	12.61s	12.65s	12.57s	1.03s	UNSAT	383	826
c1355_tc_3_40	23.04s	23.1s	22.93s	1.84s	UNSAT	546	1133
c1908_tc_3_45	1m 17s	1m 18s	1m 17s	3.67s	UNSAT	880	1793
c1908_tc_3_41	2m 33s	2m 30s	2m 26s	5.57s	UNSAT	880	1793
c1908_tc_2_43	3m 26s	3m 23s	3m 21s	11.15s	UNSAT	880	1793
c2670_tc_3_59	3m	2m 58s	2m 57s	7.53s	UNSAT	1193	2619
c2670_tc_3_55	2m 50s	2m 52s	2m 48s	8.09s	UNSAT	1193	2619
c2670_tc_3_52	2m 59s	3m 1s	3m 1s	7.63s	UNSAT	1193	2619
c2670_tc_2_58	3m 17s	3m 18s	3m 20s	7.98s	UNSAT	1193	2619
c2670_tc_2_53	3m 50s	3m 48s	3m 47s	9.34s	UNSAT	1193	2619

5. Testing, optimizations, and performance

Table 5.3.: A selection of test cases from test set **TS₃** with a performance comparison after each optimization step. In addition, the array size and the number of variables are listed.

Testcase	solution 1	solution 2	solution 3	solution 4	result	# array	# var.
c5315_tc_1_71	8.10s	8.21s	8.21s	8.69s	SAT	2307	4792
c5315_tc_1_72	9.25s	8.73s	8.73s	9.63s	SAT	2307	4792
c5315_tc_1_73	8.19s	9.88s	9.88s	7.91s	SAT	2307	4792
c5315_tc_1_74	9.71s	9.66s	9.66s	10.03s	SAT	2307	4792
c3540_tc_2_61	8m 35s	8m 33s	8m 33s	15.56s	UNSAT	1669	3388
c3540_tc_2_63	8m 59s	8m 54s	8m 54s	16.27s	UNSAT	1669	3388
c3540_tc_3_61	7m 51s	7m 55s	7m 55s	11.97s	UNSAT	1669	3388
c3540_tc_3_63	9m 20s	9m 19s	9m 19s	13.29s	UNSAT	1669	3388
c3540_tc_3_66	10m 5s	10m 1s	10m 1s	16.23s	UNSAT	1669	3388
c5315_tc_2_71	19m 58s	19m 54s	19m 54s	22.73s	UNSAT	2307	4792
c5315_tc_2_72	23m 4s	23m 10s	23m 10s	33.02s	UNSAT	2307	4792
c5315_tc_2_73	20m 33s	21m 25s	21m 25s	25.20s	UNSAT	2307	4792
c5315_tc_2_74	21m 18s	21m 6s	21m 6s	24.31s	UNSAT	2307	4792
c5315_tc_2_75	20m 25s	20m 5s	20m 5s	29.16s	UNSAT	2307	4792
c5315_tc_2_78	20m 3s	19m 57s	19m 57s	18.19s	UNSAT	2307	4792
c5315_tc_2_79	24m 2s	23m 56s	23m 56s	27.64s	UNSAT	2307	4792

Table 5.4.: A selection of test cases from test set **TS₃** having different execution times. All test cases contain 4864 variables and one array with 2416 elements.

Testcase	duration	result
c6288_tc_1_81	9.32s	SAT
c6288_tc_1_82	9.56s	SAT
c6288_tc_1_90	9.05s	SAT
c6288_tc_2_81	15m 44s	UNSAT
c6288_tc_2_82	17m 10s	UNSAT
c6288_tc_2_86	4m 20s	UNSAT
c6288_tc_2_88	22m 5s	UNSAT

5. Testing, optimizations, and performance

Table 5.5.: A selection of test cases from test set **TS₅** with comparing the performance before and after the optimization.

Testcase	solution 1	solutions 2 + 3	result
circuit10_10_2FAULTS_FAULTVERSION4_comparison	10.88s	450ms	SAT
circuit10_15_2FAULTS_FAULTVERSION3_comparison	1.72s	558ms	SAT
circuit10_15_2FAULTS_FAULTVERSION4_comparison	13.12s	514ms	SAT
circuit10_15_2FAULTS_FAULTVERSION5_comparison	17.67s	611ms	SAT
circuit10_2_1FAULTS_FAULTVERSION3_dependency	450ms	177ms	SAT
circuit10_2_1FAULTS_FAULTVERSION4_comparison	3.06s	235ms	SAT
circuit15_20_2FAULTS_FAULTVERSION3_comparison	1m 12s	1.42s	SAT
circuit15_2_2FAULTS_FAULTVERSION1_comparison	5.15s	387ms	SAT
circuit15_2_3FAULTS_FAULTVERSION2_comparison	7.19s	423ms	SAT
circuit20_15_3FAULTS_FAULTVERSION3_comparison	1m 10s	1.83s	SAT
circuit2_3_1FAULTS_FAULTVERSION4_dependency	107ms	64ms	SAT
AFW_parabola_1Faults_Fault1_dependency	17.76s	329ms	UNSAT
AFW_parabola_1Faults_Fault1_comparison	timeout 3m	551ms	UNSAT
AFW_parabola_2Faults_Fault1_comparison	timeout 3m	553ms	UNSAT
AFW_parabola_2Faults_Fault2_comparison	timeout 3m	553ms	UNSAT
AFW_parabola_2Faults_Fault3_comparison	timeout 3m	556ms	UNSAT
AFW_parabola_3Faults_Fault1_comparison	timeout 3m	558ms	UNSAT
AFW_ranking_1Faults_Fault2_comparison	1.27s	204ms	UNSAT
AFW_shopping_bedroom2_1Faults_Fault4_comparison	17.35s	330ms	UNSAT
AFW_training_1Faults_Fault5_comparison	6.03s	207ms	UNSAT
AFW_training_2Faults_Fault3_comparison	7.07s	207ms	UNSAT
AFW_weather_1Faults_Fault1_comparison	11.04s	243ms	UNSAT
AFW_weather_1Faults_Fault2_comparison	11.17s	244ms	UNSAT
AFW_weather_2Faults_Fault1_comparison	33.54s	254ms	UNSAT
AFW_weather_2Faults_Fault3_comparison	36.14s	254ms	UNSAT
AFW_wimbledon2012_2Faults_Fault2_dependency	10.87s	316ms	UNSAT
circuit10_10_2FAULTS_FAULTVERSION5_comparison	56.38s	248ms	UNSAT
circuit10_10_3FAULTS_FAULTVERSION5_comparison	22.23s	276ms	UNSAT

6. Conclusion and future work

In this work, we discussed the role of compilers in today's industrial and academic applications. Compiler technology is not only used for implementing high-level programming languages or optimization for computer architectures. They are also utilized to translate easy-to-read languages into more complex languages. [Aho+06] The same idea was pursued by [MIN2SMT](#), introduced in this thesis. This compiler translated MINION constraints into [SMT-LIB2](#) models.

To gain a better understanding of the background of the underlying technologies, Constraint Satisfiability Problems ([CSP](#)) as well as Boolean Satisfiability Problems ([SAT](#)) and Satisfiability Modulo Theories ([SMT](#)) were assessed more closely in [chapter 2](#). Their characteristics were shown, and the definitions behind them were listed.

[Chapter 3](#) addressed the structure and functioning of compilers. As the first part of a compiler, the lexical analyzer divides the input source code into smaller basic blocks, the tokens. The parser then receives these. The parser, in turn, analyzes the sequence of tokens using a set of rules to generate a syntax tree. This tree can be used to perform additional tasks such as type checking, variable verification, and enforcing other constraints on the semantics of a language. The intermediate code is generated from the syntax tree, on which optimizations can be performed. Finally, the conversion into the target language is done by the code generator.

This theoretical background was used to develop [MIN2SMT](#). This compiler was developed using the parser generator [ANTLR](#) together with the programming language Python. [Chapter 4](#) was dedicated to the architecture of this compiler and the translation of the individual MINION constraints into [SMT-LIB2](#), which the [Z3](#) solver can solve.

In [chapter 5](#), we presented the used testing techniques. In addition to unit tests developed using the *Test Driven Development* method, an integration test framework was also introduced to verify the *equivalence* between the MINION input and the [SMT-LIB2](#) output. In order to show equivalence, we distinguish two cases:

1. MINION and the generated output produce the result [UNSAT](#);
2. MINION yields one or more solutions and [Z3](#) yields also [SAT](#) whe the variables are asserted with the MINION models.

For the integration tests, multiple test sets were used. This test data was described in [section 5.2.1](#). In the course of development, some optimizations were also carried out. [Section 5.3](#) listed all implemented optimizations:

- Removal of unused variables,
- Geq/Leq optimization,
- Sum Constraints,
- Table Constraints.

This chapter concluded with a performance discussion. We found out that most of the carried-out optimizations significantly improved the execution speed of the generated code. However, in some cases, the optimizations did not have any notable effect or even led to side effects, such as higher compile time or code length.

To sum up, [MIN2SMT](#) is a suitable approach for the compilation of MINION constraint models into [SMT-LIB2](#) models.

For future work, multiple improvements in the optimizations are planned. We have seen that the implemented optimizations already have a notable influence on various aspects of performance. However, these aspects can be further improved and extended. Especially in terms of compile time and execution time in the [UNSAT](#) cases, there is still much potential.

Further improvements are also possible in the field of intermediate code representation. Since an intermediate representation of [SMT-LIB2](#) was introduced, this code could be rewritten to access the *z3-solver* library for Python [[DB23](#)]. This extension provides many possibilities to access the [Z3](#) solver directly and to perform further optimizations. Furthermore, this Intermediate Code Representation can be used to design a custom interface

that facilitates programming with [SMT-LIB2](#) and allows a combination with Python.

Another next step in this research is the integration of further programming languages, such as Prolog. The compiler's architecture is designed so that such an extension is straightforward to implement since only one additional translation function needs to be implemented per MINION language element, analogous to the `to_smtlib2()` method.

7. Abbreviations

ANTLR	ANother Tool for Language Recognition
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CSP	Constraint Solving Problem
GAC	Generalized Arc Consistency
MIN₂SMT	MINION to SMT-LIB2 compiler
SAT	Boolean satisfiability problem or satisfiable
SMT	Satisfiability Modulo Theories
SMT-LIB₂	Satisfiability Modulo Theories Library, version 2
TAC	Three Address Code
TDD	Test Driven Development
UNSAT	unsatisfiable
Z₃	SMT solver which accepts the SMT-LIB2 format

Appendix

Appendix A.

Supplementary code listings

Listing A.1: The full ANTLR4 language definition of MINION used by SMT-LIB2.

```
1 grammar Minion;
2
3 WS : [ \r\n\t ] + -> channel (HIDDEN);
4
5 COMMENT : '#' ~[\n\r]* -> channel(HIDDEN);
6
7 fragment NON_ZERO_DIGIT: '1'..'9';
8 fragment DIGIT: '0'..'9';
9 fragment LETTER : LOWERCASE | UPPERCASE;
10 fragment UPPERCASE : 'A'..'Z';
11 fragment LOWERCASE : 'a'..'z';
12
13 INT : '0' | '-'? NON_ZERO_DIGIT DIGIT*;
14
15 // different sections start keywords
16 KEY_MINION: 'MINION 3';
17 KEY_EOF: '**EOF**' .*?;
18 KEY_VARIABLES: '**VARIABLES**';
19 KEY_SEARCH: '**SEARCH**';
20 KEY_CONSTRAINTS: '**CONSTRAINTS**';
21 KEY_TUPLELIST: '**TUPLELIST**';
22
23 // VARIABLES section keywords
24 KEY_BOOL: 'BOOL';
25 KEY_BOUND: 'BOUND';
```

```

26 KEY_SPARSEBOUND: 'SPARSEBOUND';
27 KEY_DISCRETE: 'DISCRETE';
28 KEY_NOT: '!';
29
30 // SEARCH section keywords
31 KEY_VARORDER: 'VARORDER';
32 KEY_VALORDER: 'VALORDER';
33 KEY_AUX: 'AUX';
34
35 KEY_ORDER_STATIC: 'STATIC';
36 KEY_ORDER_SDF: 'SDF';
37 KEY_ORDER_SRF: 'SRF';
38 KEY_ORDER_LDF: 'LDF';
39 KEY_ORDER_ORIGINAL: 'ORIGINAL';
40 KEY_ORDER_WDEG: 'WDEG';
41 KEY_ORDER_CONFLICT: 'CONFLICT';
42 KEY_ORDER_DOMOVERWDEG: 'DOMOVERWDEG';
43 KEY_MINIMISING: 'MINIMISING';
44 KEY_MAXIMISING: 'MAXIMISING';
45 KEY_PRINT: 'PRINT';
46 KEY_ALL: 'ALL';
47 KEY_NONE: 'NONE';
48
49 ID : (LETTER | DIGIT | '_' | '-')+;
50 KEY_ASCENDING: 'a';
51 KEY_DESCENDING: 'd';
52
53
54 program : KEY_MINION input_section+ KEY_EOF;
55
56 input_section :
57     variablesSection
58     | searchSection
59     | constraintsSection
60     | tuplelistSection;
61
62 // VARIABLES section
63 variablesSection: KEY_VARIABLES varDeclaration*;
64

```

Appendix A. Supplementary code listings

```
65 domain: '{' lower=INT '..' upper=INT '}' ;
66 domainSparsebound: '{' items+=INT (',' items+=INT)* '}' ;
67
68 listDeclaration: '[' items+=INT (',' items+=INT)* ']' ;
69
70 varDeclaration: type=KEY_BOOL name=ID listDeclaration? #BoolVarDecl
71                | type=KEY_SPARSEBOUND name=ID listDeclaration?
72                  domainSparsebound #SparseboundVarDecl
73                | type=KEY_BOUND name=ID listDeclaration? domain
74                  #BoundVarDecl
75                | type=KEY_DISCRETE name=ID listDeclaration? domain
76                  #DiscreteVarDecl
77                ;
78
79 // CONSTRAINTS section
80
81 constraintsSection: KEY_CONSTRAINTS constraint* ;
82
83 constraint: name=ID '(' arg (',' arg)* ')' #SimpleConstraint
84            | name=ID '(' '{' constraint (',' constraint)+ '}' ')'
85              #WatchedConstraint
86            ;
87
88 tableConstraint: '{' rows+=tableRow (',' rows+=tableRow)* '}' ;
89 tableRow: '<' cols+=INT (',' cols+=INT)* '>' ;
90
91 arg: KEY_NOT? value=ID #ArgVariable
92     | KEY_NOT? value=INT #ArgInteger
93     | constraint #ArgConstraint
94     | list #ArgList
95     | listAccess #NotUsedArgListAccess
96     | tableConstraint #ArgTableConstraint
97     ;
98
99 list: '[' value+=listValue (',' value+=listValue)* ']' ;
100 listValue: KEY_NOT? value=ID #ListVariable
101           | KEY_NOT? value=INT #ListInteger
102           | listAccess #NotUsedListAccess
103           | list #NotUsedList
```

Appendix A. Supplementary code listings

```
100      ;
101 listAccess: KEY_NOT? name=ID list;
102
103 // SEARCH section
104
105 searchSection: KEY_SEARCH varValOrdering* optimisationFn?
106     printFormat?;
107
108 varValOrdering: varOrder valOrder?;
109
110 order: KEY_ORDER_STATIC
111     | KEY_ORDER_SDF
112     | KEY_ORDER_SRF
113     | KEY_ORDER_LDF
114     | KEY_ORDER_ORIGINAL
115     | KEY_ORDER_WDEG
116     | KEY_ORDER_CONFLICT
117     | KEY_ORDER_DOMOVERWDEG;
118
119 varOrder: KEY_VARORDER KEY_AUX? order? list;
120
121 valOrder: KEY_VALORDER '[' (KEY_ASCENDING | KEY_DESCENDING)+ '];
122
123 optimisationFn: (KEY_MINIMISING | KEY_MAXIMISING) ID;
124
125 printFormat: KEY_PRINT (list | KEY_ALL | KEY_NONE);
126
127 tuplelistSection: KEY_TUPLELIST tupleList*;
128 tupleList: name=ID rows=INT cols=INT numbers+=INT+;
```

Bibliography

- [Aho+06] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Aug. 2006. ISBN: 0321486811. URL: <http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20%5C&path=ASIN/0321486811> (cit. on pp. 14–16, 20, 65, 75).
- [Beco2] Kent Beck. *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam, 2002. ISBN: 0321146530 (cit. on p. 57).
- [Bol+09] Carl Friedrich Bolz et al. “Tracing the Meta-Level: PyPy’s Tracing JIT Compiler.” In: *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. IC00OLPS ’09. Genova, Italy: Association for Computing Machinery, 2009, pp. 18–25. ISBN: 9781605585413. DOI: 10.1145/1565824.1565827. URL: <https://doi.org/10.1145/1565824.1565827> (cit. on p. 23).
- [BPS99] Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. “Constraint satisfaction problems: Algorithms and applications.” In: *European Journal of Operational Research* 119.3 (1999), pp. 557–581. ISSN: 0377-2217. DOI: [https://doi.org/10.1016/S0377-2217\(98\)00364-6](https://doi.org/10.1016/S0377-2217(98)00364-6). URL: <https://www.sciencedirect.com/science/article/pii/S0377221798003646> (cit. on pp. 5, 6).
- [BSV10] Miquel Bofill, Josep Suy, and Mateu Villaret. “A System for Solving Constraint Satisfaction Problems with SMT.” In: July 2010, pp. 300–305. ISBN: 978-3-642-14185-0. DOI: 10.1007/978-3-642-14186-7_25 (cit. on p. 1).
- [Cor23] David Cortesi. *pyinstaller*. Website. Apr. 2023. URL: <https://pyinstaller.org/> (cit. on p. 24).

- [Dav13] Inc. David R. Cok GrammaTech. *The SMT-LIBv2 Language and Tools: A Tutorial*. Mar. 2013 (cit. on pp. 9, 54).
- [DB23] Audrey Dutcher and Nikolaj Bjorner. *z3-solver*. Website. May 2023. URL: <https://pypi.org/project/z3-solver/> (cit. on p. 76).
- [DGN10] Luc De Raedt, Tias Guns, and Siegfried Nijssen. “Constraint Programming for Data Mining and Machine Learning.” In: vol. 3. Jan. 2010 (cit. on p. 4).
- [Elg15] Hani Abdalla Muftah Elgabou. “Encoding The Lexicographic Ordering Constraint in Satisfiability Modulo Theories.” Master’s Thesis. University of York, Feb. 2015 (cit. on pp. 8–10).
- [Fra12] Josep Suy Franch. “A Satisfiability Modulo Theories Approach to Constraint Programming.” PhD Thesis. Universitat de Girona, 2012 (cit. on pp. 1, 8).
- [Frü23] Stephan Frühwirth. *MINION to SMT-LIB2 Compiler*. Git Repository. May 2023. URL: <https://gitlab.com/master-thesis-fruehwirt/minion-to-smt-lib2-compiler> (cit. on p. 1).
- [GJM06] Ian P. Gent, Chris Jefferson, and Ian Miguel. “MINION: A Fast, Scalable, Constraint Solver.” In: *Proceedings of the 2006 Conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva Del Garda, Italy*. NLD: IOS Press, 2006, pp. 98–102. ISBN: 1586036424 (cit. on pp. 1, 6, 7).
- [Hof+] Birgit Hofer et al. “Choosing Abstraction Levels for Model-based Software Debugging: A Theoretical and Empirical Analysis for Spreadsheet Programs.” unpublished. (Cit. on pp. 1, 60).
- [Hof14] Birgit Hofer. “Spectrum-Based Fault Localization for Spreadsheets: Influence of Correct Output Cells on the Fault Localization Quality.” In: *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. 2014, pp. 263–268. DOI: [10.1109/ISSREW.2014.100](https://doi.org/10.1109/ISSREW.2014.100) (cit. on pp. 1, 60).
- [Hof15] Georg Hofferek. *Logic and Computability*. Lecture Notes. Institute of Applied Information Processing and Communications (IAIK), Graz University of Technology. 2015 (cit. on p. 8).

- [Höf15] Andrea Höfler. “On the Usage of Value- and Dependency-based Models for Spreadsheet Debugging with SMT Solvers.” Master’s Thesis. Institute of Software Technology (IST), Graz University of Technology, 2015 (cit. on pp. 4, 6, 8–10, 12, 13, 60).
- [Hof18] Birgit Hofer. *Selected Topics Softwaretechnology* 3. Lecture Notes. Institute of Software Technology (IST), Graz University of Technology. Jan. 2018 (cit. on pp. 18, 20, 61, 64).
- [Hof23] Birgit Hofer. *Model Evaluation - Supplemental material*. Git Repository. Mar. 2023. URL: <https://github.com/bhoferTU/Abstraction-Levels-for-Model-based-Software-Debugging> (cit. on p. 61).
- [Ini23] The SMT-LIB Initiative. *SMT-LIB Website*. Website. May 2023. URL: <http://smtlib.cs.uiowa.edu/> (cit. on pp. 1, 11).
- [JBP19] Willy Jordan, Agus Bejo, and Anugerah Galang Persada. “The Development of Lexer and Parser as Parts of Compiler for GAMA32 Processor’s Instruction-set using Python.” In: *2019 International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*. 2019, pp. 450–455. DOI: [10.1109/ISRITI48646.2019.9034617](https://doi.org/10.1109/ISRITI48646.2019.9034617) (cit. on p. 23).
- [Jef+14] Christopher Jefferson et al. *The Minion Manual*. July 2014 (cit. on pp. 1, 6, 31, 35, 42, 60, 62).
- [Koi+19] Roxane Koitz-Hristov et al. *Compiler Construction*. Lecture Notes. Institute of Software Technology (IST), Graz University of Technology. 2019 (cit. on pp. 14, 15, 19, 21).
- [MBo9] Leonardo de Moura and Nikolaj Bjørner. “Satisfiability Modulo Theories: An Appetizer.” In: *Formal Methods: Foundations and Applications*. Ed. by Marcel Vinícius Medeiros Oliveira and Jim Woodcock. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 23–36. ISBN: 978-3-642-10452-7 (cit. on pp. 9, 10).
- [Par23a] Terence Parr. *ANTLR*. Website. Apr. 2023. URL: <https://www.antlr.org/> (cit. on p. 20).
- [Par23b] Terence Parr. *antlr4-tools*. Website. Apr. 2023. URL: <https://github.com/antlr/antlr4-tools> (cit. on p. 24).

- [Pug04] Jean-Francois Puget. “Constraint Programming Next Challenge: Simplicity of Use.” In: *Principles and Practice of Constraint Programming – CP 2004*. Ed. by Mark Wallace. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 5–8. ISBN: 978-3-540-30201-8 (cit. on p. 6).
- [Res23a] Microsoft Research. *Z3 - Project Website*. Website. May 2023. URL: <https://www.microsoft.com/en-us/research/project/z3-3/> (cit. on pp. 2, 10).
- [Res23b] Microsoft Research. *Z3 - the theorem prover*. Git Repository. May 2023. URL: <https://github.com/z3prover/z3> (cit. on p. 12).
- [Rya10] Malcolm Ryan. “Constraint-based multi-robot path planning.” In: *2010 IEEE International Conference on Robotics and Automation*. 2010, pp. 922–928. DOI: [10.1109/ROBOT.2010.5509582](https://doi.org/10.1109/ROBOT.2010.5509582) (cit. on p. 4).
- [Sim01] Helmut Simonis. “Building Industrial Applications with Constraint Programming.” In: *Constraints in Computational Logics: Theory and Applications*. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 271–309. ISBN: 3540419500 (cit. on pp. 4, 6).
- [Tea23] The PyPy Team. *PyPy Website*. Website. Apr. 2023. URL: <https://www.pypy.org/> (cit. on p. 23).
- [VPH23] Eric Vergnaud, Terence Parr, and Sam Harwell. *antlr4-python3-runtime*. Website. Apr. 2023. URL: <https://pypi.org/project/antlr4-python3-runtime/> (cit. on p. 24).
- [Wal00] Toby Walsh. “SAT v CSP.” In: *Principles and Practice of Constraint Programming – CP 2000*. Ed. by Rina Dechter. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 441–456. ISBN: 978-3-540-45349-9 (cit. on p. 9).
- [Wal23] Jacob Walls. *pylint*. Website. Apr. 2023. URL: <https://pylint.readthedocs.io/> (cit. on p. 24).
- [WNA10] Franz Wotawa, Mihai Nica, and Bernhard K. Aichernig. “Generating Distinguishing Tests Using the Minion Constraint Solver.” In: *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. 2010, pp. 325–330. DOI: [10.1109/ICSTW.2010.11](https://doi.org/10.1109/ICSTW.2010.11) (cit. on pp. 1, 60).