MIN2SMT – A MINION to SMT-LIB2 Compiler

Stephan Frühwirt
Graz University of Technology, Institute of Software Technology, Inffeldgasse 16b/2, Graz, Austria

Roxane Koitz-Hristov
Graz University of Technology, Institute of Software Technology, Inffeldgasse 16b/2, Graz, Austria

Franz Wotawa[1]
Graz University of Technology, Institute of Software Technology, Inffeldgasse 16b/2, Graz, Austria

Abstract

Constraint satisfaction solvers are fundamental tools for addressing diverse real-world challenges; however, different solvers often require models written in distinct programming or modeling languages, making interoperability and performance comparison difficult. To partially solve this challenge, we introduce a compiler that takes input models written for the MINION constraint solver and converts them to equivalent SMT-LIB2 representations. Our compiler is publicly available. Furthermore, we present a testing methodology to verify the correctness of our translations and empirically evaluate our compiler using the solver Z3. In this experiment, we not only tested the conversions of basic MINION language elements but also considered complex MINION models used for diagnosis.

2012 ACM Subject Classification Software and its engineering → Compilers; Computing methodologies → Artificial intelligence; Computing methodologies → Representation of mathematical objects; General and reference → Verification

Keywords and phrases Constraint modeling, model compilation, evaluation of model compilers

1 Introduction

Constraint satisfaction problems (CSPs) are the basis for many solutions to real-world problems such as configuration [24] and diagnosis [4]. The underlying idea is to formulate a problem as a CSP and to use a constraint solver to compute solutions. Due to the advancements in CSP solving, we can quickly obtain solutions to specific problems using general solvers such as MINION [17, 20], Choco [26], or Z3 [13]. For example, in previous research on software fault localization, the authors used MINION [29] and Z3 [6] to compute the root causes of detected failures of programs. There, the underlying concept was to map programs into a constraint representation. Unfortunately, many CSP solvers come with specific input languages requiring to adapt the constraint representation accordingly, causing additional effort. This mentioned problem motivates developing a compiler that allows us to reuse CSP representations originally developed for one solver to be converted into a representation for another.

A compiler that maps CSPs written in one solver language into another has several applications. We discuss them focusing on MINION and the SMT-LIB2 [10] used by Z3 and other solvers. A compiler from MINION to SMT-LIB2 utilizes *interoperability and model reuse*. The integration of MINION and SMT-LIB2 promotes the interoperability between different constraint modeling and solving frameworks. It also facilitates the exchange of models and solutions. This is especially relevant as a *Research Tool* where the compilation of MINION constraints to the SMT-LIB2 format facilitates research on solving algorithms by automating the translation of benchmarks written in MINION constraints for comparing solver implementations using at least similar examples. It is worth noting

[1] Corresponding author

that the compiled constraints might not be the most efficient. However, given the translated number of constraints in SMT-LIB2 format is polynomial-bounded, we do not add additional computational complexity for solving.

The application of a compiler also supports *tool dissemination and its ecosystem*. Many constraint solvers understand the SMT-LIB2 format, such as Z3. We focus on Z3 as it is a widely adopted SMT solver that is used and supported by research and industrial communities. By compiling MINION constraints to SMT-LIB2 for solvers such as Z3, seamless integration with these tools and workflows is possible. This minimizes the need for custom interfaces or data conversions. Moreover, we can rely on different solvers without having to manually convert available MINION models.

The compiler may also lead to *performance enhancements*. By compiling MINION constraints to SMT-LIB2 models, one can benefit from the advanced solving techniques of solvers such as Z3, potentially leading to faster and more scalable solutions for MINION-based CSPs. For instance, while MINION is designed to exploit modern hardware architectures, Z3 further includes learning capabilities improving its performance.

Finally, the compiler can be used for *verification and validation* of solvers. Converting models from one language to another supports testing activities. We can test the output of constraint solvers like MINION and Z3 using the same but differently coded constraint problem. Given the correctness of the conversion, we can solve the same constraint problem using MINION and Z3 and look at the obtained solutions.

Similarly to our work, Bofill et al. [9] present FZN2SMT. FZN2SMT compiles FlatZinc, which is an intermediate code for the MiniZinc constraint modeling language, to SMT-LIB and automatically determines the suitable, i.e., simplest, logic during translation. Their empirical evaluation demonstrates that SMT can enhance the efficiency and scalability of CSP solutions. FZN2OMT [11] is a framework that converts FlatZinc/MiniZinc into suitable models for Optimization Modulo Theories (OMT), an extension of SMT, and vice versa. While the tool can be readily integrated with the MiniZinc toolchain, the authors encountered performance issues in regard to the generated models. It is worth noting that besides compiling one modeling language for CSPs into another, there are also other ideas to overcome the problem of different input modeling languages. One, for example, is to introduce solver-independent languages like Essence/Essence Prime [1, 2]. Interestingly, there is work on mapping Essence Prime into a MINION representation. These tools aid in constraint modeling by converting constraint problem models formulated in Essence Prime into the input format of the MINION CSP solver [21] or SMT-LIB [12].

Our proposed MINION to SMT-LIB2 compiler MIN2SMT[2] takes MINION input models and converts them to an equivalent SMT-LIB2 representation. We tested the conversion on a large number of models and checked for equivalence of results when calling MINION and Z3 on the original and the compiled model. In this paper, we summarize the basic principles of the compilation and present the testing methodology used. We further discuss current limitations. Note that the compiler is available for free, including the source code [3].

We structure the paper as follows: First, we discuss the basic foundations, the underlying solvers, and the principles behind testing. Afterward, we discuss the compilation methodology used, followed by a detailed evaluation and testing section, where we also introduce

[2] Portions of this work have been previously published as part of the Master thesis [16] of one of the authors.
[3] https://gitlab.com/master-thesis-fruehwirt/minion-to-smt-lib2-compiler

implemented optimizations. Finally, we conclude the paper.

2 Foundations

Formulating problems, such as diagnosis or configuration, in the form of constraints and using a solver for computing solutions have been active research areas for decades. Several papers and introductory books deal with the corresponding CSP, e.g., Dechter [14]. In this section, we summarize the foundations and available tools. We start by defining a CSP. For illustration purposes, we use *the farmer's Problem* from [20]:

"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?"
To solve this problem, we first have to formulate it as a CSP.

▶ **Definition 1** (Constraint Satisfaction Problem (CSP)). *A Constraint Satisfaction Problem (CSP) is a triple* (V, D, CO) *where*

- V *is a finite set of variables* $v_1, \ldots, v_n$
- D *is a finite set of domains* $d_1, \ldots, d_n$ *for each variable. Each* d_i *specifies the value a variable* v_i *can take.*
- CO *is a finite set of constraints* $c_1, \ldots, c_k$*, were each constraint* c_i *is a relation between a set of variables* $S_i \subseteq V$*, which is called the scope of constraint* c_i*. Each relation itself is a set of tuples a variable can take.*

Note that in Definition 1, we assume a relation to be defined as a set of tuples. In practice, we might not define such a relation by stating all possible tuples. Instead, we assume relations and operations like $<$ or $\neg$ that implicitly define such a tuple space. A constraint can be fulfilled or violated. Before defining fulfillment or violation, we first introduce the concept of value assignments.

Using the CSP definition, we formalize the farmer's problem as follows:

$$FP = (\{p, h\}, \{p, h \in \mathbb{N}_0\}, \{p + h = 7, 4 \cdot p + 2 \cdot h = 22\})$$

In the CSP FP p represents the number of pigs, and h the number of hens. Both variables are natural numbers. To solve the CSP, we need to assign values to variables such that all constraints are fulfilled. Formally, we start describing value assignments.

▶ **Definition 2** (Value assignment). *Given a CSP* (V, D, CO)*. A value assignment is a set of tuples* (v_i, x_i) *where* $v_i \in V$*, and* $x_i \in d_i$ *where* $d_i \in D$ *is the domain of the corresponding variable* v_i*. Note that we assume that there is exactly one value for each variable in a value assignment.*

Given a constraint c from a CSP (V, D, CO) and a value assignment Γ, we define constraint fulfillment and violation as follows:

▶ **Definition 3** (Constraint fulfillment/violation). *The value assignment* Γ *fulfills a constraint* c *with scope* $\{v_1, \ldots, v_m\}$ *if and only if* $(x_1, \ldots, x_m)$ *with* $(v_i, x_i) \in \Gamma$ *is in the relation of the constraint. Otherwise, we say that the value assignment violates the constraint.*

A solution to the CSP is a value assignment that does not violate any constraint.

▶ **Definition 4** (CSP solution). *A value assignment* Γ *for a CSP* (V, D, CO) *is a solution if and only if it does not violate any constraint* $c \in CO$*.*

For the CSP FP formalizing the farmer's problem, a solution is:

$$p = 4, h = 3$$

Note that we may have not only one solution but many of them. Depending on the constraint solver and parameters, we may obtain one or all solutions.

To solve a given CSP, every constraint solver searches for a solution that considers the constraints and the variables' domains. It is worth noting that constraint solving can be seen as an extension to SAT solving [18], where we only consider Boolean domains and Boolean operators as constraints. For more information regarding solving techniques, we refer to Dechter [14]. In this paper, we assume that we have a solver S that provides us with solutions for a given CSP. In particular, we rely on the MINION constraint solver and Z3. MINION is a fast and scalable constraint solver that supports a wide variety of constraints. The syntax of MINION constraints is similar to the syntax of function calls in various high-level languages such as C. In addition to constraints for modeling Boolean properties such as equality, disequality, and inequality, constraints for modeling arithmetic problems such as sum or product are provided. There are also constraints for describing tables [17]. Z3 [27] is an SMT solver that allows reasoning over various mathematical structures combined with a Boolean SAT solver. Developed by Microsoft Research, Z3 is a high-performance SMT solver that supports a wide range of theories, including arithmetic, arrays, bit-vectors, and quantifiers. One of the key features of Z3 is its ability to handle complex formulas and theories. It can solve formulas that involve multiple theories, as well as handle quantifiers, which are often used in program verification and optimization. Additionally, the Z3 API enables interaction with Z3 from other programming languages, including Python, Java, and C++.

When using a constraint solver, we have to formulate the CSP in the appropriate modeling languages, which come with their particularities. For example, in Listing 1 and Listing 2, we formulate the farmer's problem in MINION[4] and SMT-LIB2, respectively.

Listing 1 The farmers problem in the MINION input language.

```
MINION 3

**VARIABLES**
DISCRETE pigs {0..7}
DISCRETE hens {0..7}

**CONSTRAINTS**

weightedsumgeq([2,4], [hens,pigs], 22)
weightedsumleq([2,4], [hens,pigs], 22)
sumgeq([hens,pigs],7)
sumleq([hens,pigs],7)

**EOF**
```

Listing 2 The farmers problem formalized in SMT-LIB2.

```
(declare-const pigs Int)
(declare-const hens Int)
```

[4] Note that there is no equality constraint for weighted sum and sum in MINION.

```

(assert (<= 0 pigs 7))
(assert (<= 0 hens 7))
(assert (= (+ (* 2 hens) (* 4 pigs)) 22))
(assert (= (+ hens pigs) 7))

(check-sat)
```

Both representations of the farmer's problem CSP FP are rather different. To compare two different CSP solvers, we must develop models for both separately, which requires additional effort. Hence, a compiler that takes a CSP written in the modeling language of one CSP solver and converts it into a modeling language the second CSP solver can process is an effective way to reduce this effort. However, to be of use, we must ensure that both CSP solvers compute the same values for the original and the compiled model, respectively, i.e., that the compiler works as expected.

We can ensure the correctness of the compiler in two ways. We may want to verify the compiler formally. Formal verification can be obtained by proving that every mapping from one constraint in language L_1 into constraints in language L_2 is correct. For this part, we only need to show that both representations are feasible for the same inputs. In addition, we would need to check that the variable conversion is correct. Note that formally showing model equivalence is difficult to achieve [22]. The second way of ensuring correctness is testing [23]. For this purpose, we select models $m_1, \ldots, m_n$ written in language L_1, compile them into $\gamma(m_1), \ldots, \gamma(m_n)$ in language L_2, where γ is the translation function. Afterwards, check the computed outcome of CSP solver C_1 and C_2. In case that for all $i = 1 \ldots n$, both solvers compute the same output, i.e., $C_1(m_i) = C_2(\gamma(m_i))$, the compilation is correct (at least for the provided models). Note that this testing technique is ambiguous. Do we assume that both constraint solvers deliver all solutions, or is it sufficient that they can distinguish solutions to be solvable or unsolvable? We will clarify these questions in the concrete testing framework proposed later in this paper.

It is worth noting that formal verification does not always guarantee correctness in practice, and neither does testing. For the former, we may make assumptions about the computing environment, e.g., assuming infinite memory, which is not true in practice. Donald Knuth stated this well: *"Beware of bugs in the above code; I have only proved it correct, not tried it."* Hence, testing needs to be performed anyway.

We do not formally prove the correctness for our concrete MIN2SMT compiler. However, we provide the SMT-LIB2 representation for each MINION constraint that the compiler considers. This mapping allows us to informally assess the correctness of the constraint conversion. Furthermore, we introduce an integration testing framework for verifying the mapping. In Section 4, we outline the testing approach and the obtained results in detail.

3 Compilation methodology

This section presents the practical approach to compiling MINION models into their corresponding SMT-LIB2 encoding. Our compiler has been implemented in Python3 using the parser generator ANTLR [25]. We use the naming conventions outlined in Table 1 to simplify the explanation.

Table 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

3.1 Constraint variable declarations

MIN2SMT supports three types of MINION variables: DISCRETE, SPARSEBOUND, and BOOL. The first two types always have a detailed specification of their domain, while Boolean variables inherently have the domain $\{0, 1\}$. For example, we may set the domain of a DISCRETE type to $1 \ldots 10$ and the one for SPARSEBOUND to the values $1, 3, 10$. Variables can also appear as (multidimensional) vectors. Listing 4 depicts the conversion of the variable definitions from Listing 3 to SMT-LIB2.

Listing 3 Supported MINION constraint variables.

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

Listing 4 Tranlsation of different MINION constraint variable types to SMT-LIB2.

```
(declare-const A (Array Int Int))
(declare-const a Int)
(declare-const sb Int)
(declare-const b Bool)
(declare-const  ab (Array Int Bool))

; A[2] {-1..5}
(assert
  (forall ((i Int))
    (=> (<= 0 i 1) (<= -1 (select A i) 5))
  )
)
; a {0..40}
(assert (<= 0 a 40))
; sb {1,3,4,5}
(assert(or(= sb 1)(= sb 3)(= sb 4)(= sb 5)))
```

3.2 Constraints

In this subsection, we list s subset of the available MINION constraints with descriptions adapted from Jefferson et al. [20] and provide additional information with regard to the implementation of MIN2SMT. The full list can be found in Appendix A.

MINION constraint — **SMT-LIB2 Conversion**

`alldiff(A)` ensures that each element of A takes a different value.

```
(assert
  (forall ((i Int) (j Int))
    (=>
      (and
        (< 0 i n)
        (< 0 j n)
        (= (select A i) (select A j))
      )
      (= i j)
    )
  )
)
```

`div(x, y, z)` ensures that $\lfloor \frac{x}{y} \rfloor = z$ and is always **`false`** in case of $y = 0$. The MINION implementation of the division differs from the standard implementation of the division in SMT-LIB2. Jefferson et al. [20] presents the following examples: $\frac{10}{3} = 3$, $\frac{-10}{3} = -4$, $\frac{10}{-3} = -4$ and $\frac{-10}{-3} = 3$.

```
(assert
  (and
    (distinct y 0)
    (ite
      (or
        (and (> x 0) (> y 0))
        (and (< x 0) (> y 0))
      )
      (= z (div x y))
      (=
        (div (- x) (- y))
        z
      )
    )
  )
)
```

`element(A,i,e)` ensures that $A[i] = e$, where $0 \leq i \leq n$. The constraint is **`false`**, if i is outside the index range.

```
(assert
  (and
    (= (select A i) e)
    (< i n)
  )
)
```

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

```
(assert
  (>=
    (+
      (ite
         (distinct
           (select A 0) (select B 0)
         )
         1 0
       )
       ...
       (ite
         (distinct
           (select A n) (select B n)
         )
         1 0
       )
    )
    c
  )
)
```

`lexleq(A, B)` ensures that A is lexicographically less than or equal to B, where A and B are both of same length.

```
(define-fun-rec fun_lexleq ((i Int))
Bool
  (ite
    (>= i n)
    true
    (ite
      (> (select A i) (select B i))
      false
      (ite
        (< (select A i) (select B i))
        true
        (fun_lexleq (+ i 1))
      )
    )
  )
)
(assert (fun_lexleq 0))
```

`occurrence(A, c, x)` ensures that the value x occurs exactly c times in A.

```
(assert
  (=
    c
    (+
      (ite (= (select A 0) x) 1 0)
      ...
      (ite (= (select A n) x) 1 0)
    )
  )
)
```

`watchvecneq(A, B)` ensures that A and B are not the same, i.e., $\exists i : A[i] \neq B[i]$.

```
(assert
  (or
    (distinct (select A 0) (select B 0))
    ...
    (distinct (select A n) (select B n))
  )
)
```

3.3 Optimizations

Optimizations are essential in compiler construction. They enhance the generated code's performance by minimizing the number of instructions executed by the target machine while ensuring that the program's behavior remains unaltered. Besides removing unused variables that unnecessarily lengthen the translated code and would also be included in the solution-finding process, we implemented three optimizations to reduce the execution time on the compiled SMT-LIB2 encoding.

3.3.1 Geq/Leq Optimization

Due to implementation considerations, MINION does not support a "sum equals" constraint [20]. Hence, in order to create such a relation, two constraints are necessary: `sumgeq` and `sumleq`, encoding "sum greater equals" and "sum less equals" (see Listing 1 as an example). However, in MIN2SMT, we employed an optimization to replace `sumgeq` / `sumleq` pairs with the corresponding `sumeq` constraint. This optimization is only applicable if the related `sumgeq`/`sumleq` constraints occur in the input code using the exact same arguments. This optimization reduces the code size of the resulting SMT-LIB2 code, as the two constraints are replaced by a single remaining constraint, which has to be translated after the optimization. For the same reason, this optimization has been implemented for the `watchsumgeq` / `watchsumleq` and `weightedsumgeq` / `weightedsumleq` constraints.

3.3.2 Sum Constraints

Several constraints, such as `sumgeq`/ `sumleq` or `hamming`, require a summation logic when translated to SMT-LIB2. In our original summation strategy, we simply looped over all elements to compute the sum; however, the execution of many tests, had to be canceled manually, as Z3 sometimes took several hours to find a solution. Thus, it was optimized in terms of execution time by performing loop unrolling [5]. In loop unrolling the number of iterations is reduced by repeating similar independent statements instead of performing a loop. While this optimization can reduce the execution time significantly, it has the disadvantage that more instructions than in the original code are necessary. Listing 12 and 13 demonstrates the loop unrolling technique.

Listing 12 Example input of the loop unrolling optimization for the `sumgeq` constraint. `sumgeq(A, a)` ensures that $\sum_i A_i \geq a$.

```
DISCRETE A[5] {0..6}
DISCRETE a {0..6}
sumgeq(A, a)
```

Listing 13 Example output of the loop unrolling optimization for the constraint in Listing 12.

```
(assert
  (>=
    (+
        (select A 0)
        (select A 1)
        (select A 2)
        (select A 3)
        (select A 4)
    )
    a
  )
)
```

3.3.3 Table Constraints

`table` specifies a constraint via a list of tuples, such that each tuple represents the allowed assignments, i.e., `table(A, T)` ensures that there exists at least one column t in table T, such that $t = A$. Its counterpart `negativetable`, thus specifies all disallowed assignments in form of a table. These two special constraint types are defined within the `**TUPLELIST**` section of the MINION code. Listing 14 provides an example of the usage of the `table` constraint, where a 2×2 table, i.e., featuring two tuples and two variables, with the identifier A is defined. Adding the `table` constraint in line 8 ensures that the variables of vector $[a, 2]$ will satisfy the constraint A.

We decided to define tables with a fixed number of rows and columns and concrete values as this pre-initialization allows the compiler itself to already prepare the individual values accordingly. Thus, there is no need to compare entire columns anymore; instead, individual values are compared. This results in a disjunction of conjunctions of value comparisons, as shown in Listing 15. The values of the table are inserted directly in the assertion at compile time.

Listing 14 Example of the `table` constraint

```
**VARIABLES**
BOOL a
**TUPLELIST**
A 2 2
1 2
3 4
**CONSTRAINTS**
table([a, 2], A)
```

Listing 15 Optimized output for the `table` constraint from Listing 14.

```
(assert
  (or
    (and
      (= 1 (ite a 1 0))
      (= 2 2)
    )
    (and
      (= 3 (ite a 1 0))
      (= 4 2)
    )
  )
)
```

3.4 Limitations

Currently, the MINION constraints `gacschema`, `gcc`, `gccweak`, `haggisgac`, `haggisgac_stable`, `lighttable`, `mddc`, `negativemddc`, `shortstr2`, and `str2plus` are unsupported because they implement unique algorithms or address CSP-specific issues, such as Generalized Arc Consistency (GAC). GAC is 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. Furthermore, some MINION constraints that explicitly enforce GAC but are otherwise identical to other constraints were mapped to their companion constraints, e.g., `watchelement`, `watchelement_one`, and `gacalldiff`.

MIN2SMT does not support the `**SEARCH**`[5] and `**SHORTTUPLELIST**`[6] sections currently. The search section allows for the definition of a variable ordering for the output, a value ordering, and an objective function, as well as the specification of which variables should be printed. However, SMT-LIB2 does not provide any of these features [8]. The short tuple list section is not used since no constraints that accept short tuple lists were implemented.

The direct translation used by MIN2SMT for Boolean variables may lead to an incorrect SMT-LIB2 representation. In MINION summing up Boolean values can be done, whereas this is not allowed when using Z3 on the compiled SMT-LIB2 model.

4 Testing and Evaluation

Software testing is a crucial part of software development. It detects deviations from the software specification and reduces the probability of occurrence of errors in production. We developed and ran unit tests on all classes and components involved in the compilation process. 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, i.e., integration testing.

[5] In the search section the user can specify, for instance, variable orderings or details on how to print the solution output.

[6] Short tuples allow tuples to be expressed as a smaller list and are only accepted by a limited set of constraints.

4.1 Integration Testing

The main aim is to examine 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, we distinguish two cases: (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 for both cases. While the `UNSAT` case is trivial to test, in the `SAT` case the following integration test framework is necessary (see Figure 1):

1. The MINION code is run by the MINION solver, using the `-findallsols` flag. If this flag is set, all possible solutions, i.e., all solutions, will be found and listed.
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 meta-information 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.

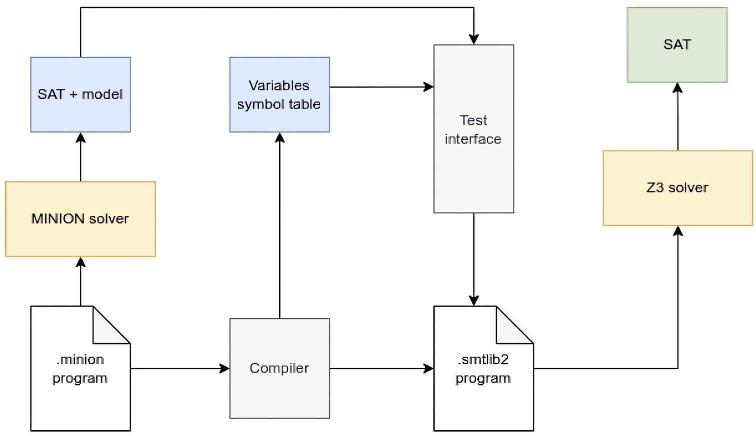


Figure 1 Workflow of the test framework in the `SAT` case.

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 in the output and in which order.

4.2 Test data

We gathered a large collection of different categories of programs to enable thorough integration testing:

TS1 Constraint test cases: For each supported MINION constraint, we have written one or multiple test files using different types of parameters and variable types.

TS2 MINION examples: This set includes various complex CSPs or logic puzzles such as *The farmer's problem*, *The Zebra Puzzle*, or the *N-Queens* problem from Jefferson et al. [20].

TS3 ISCAS85 data set: This benchmark contains combinational circuits [19] and 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 [28] utilized these test cases for experimental and benchmarking activities. The following constraints were used in these files: `watched-or`, `eq`, `ineq`, `reify`, `element`, `watchsumgeq`, `watchsumleq`, `sumgeq`, and `sumleq`.

TS5 Spreadsheet evaluation data: These MINION files were generated automatically from spreadsheets [3].

Overall, more than 1,700 test files were used for extensive integration testing and the MIN2SMT compiler passed all test cases.

4.3 Experimental evaluation

For the empirical evaluation of the compiler and its optimizations, we utilize our integration test framework and subsets of the test suites described previously. In particular, we focus on the execution speed of the generated code with and without the implemented encoding optimizations described in Section 3.3 using the SMT-solver Z3. The experiments were executed on a computer with an AMD®Ryzen 5 5625u processor (4.3 GHz, six cores) with 16 GB RAM under Ubuntu 22.04, 64-bit.

4.3.1 Geq/Leq Optimization

The Geq/Leq optimization introduced in Section 3.3.1 reduces the code size by the translation of two constraints to one. We evaluated this optimization on 38 test cases from test set TS3. In Figure 2, we depict the runtime distributions of the original and optimized encoding on all test cases, on only the `UNSAT` test cases and on only the `SAT` test cases. The original version required 347.62 (*Median*=115.00, *Standard Deviation*=483.67, *Min*=0.73, *Max*=$1,442.00$) seconds on average over all test cases while the optimization reduced this number to 27.74 (*Median*=8.39, *Standard Deviation*=108.88, *Min*=0.35, *Max*=679.00) seconds. However, the disadvantage of this optimization method is that the compile time is nearly doubled for test cases within this test suite. Furthermore, for test cases with the result `SAT`, there is hardly any difference in whether the optimization is active or not. Only in those test cases where the result is `UNSAT` and the search takes a long time the search duration is significantly reduced in most cases.

4.3.2 Sum Constraints

Twenty different randomly chosen test cases from TS3 were used for benchmarking the sum constraint optimization. The used test cases feature arrays with sizes between 383 to $2,307$ elements, including between 445 to $4,792$ variables. Our evaluation revealed that the original and optimized compilation technique had a duration of 501.36 (*Median*=47.15, *Standard Deviation*=679.74, *Min*=1.16, *Max*=1857.00) and 368.10 (*Median*=54.17, *Standard Deviation*=517.80, *Min*=1.14, *Max*=$1,491.00$) seconds on average, respectively. The optimized version achieved a noticeable performance boost on several test cases, especially in

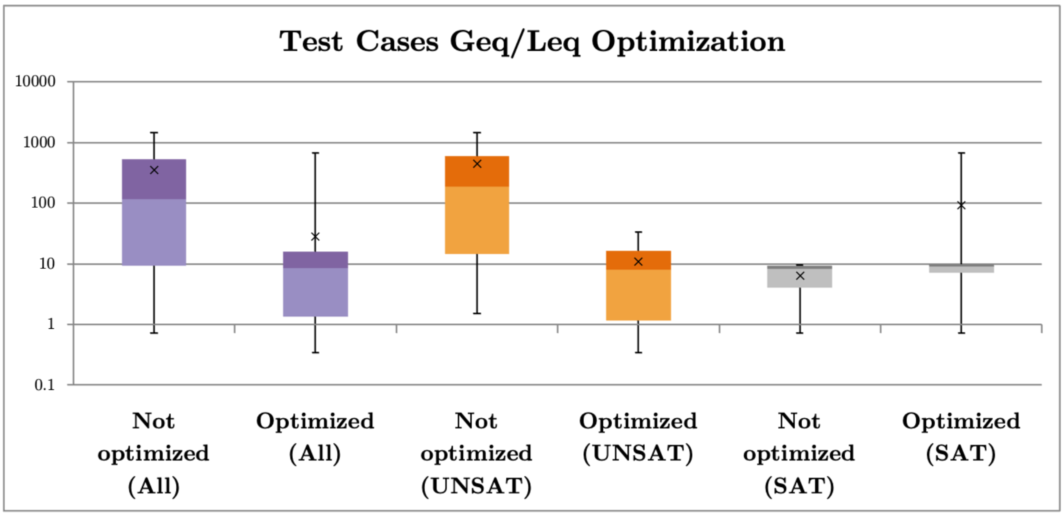
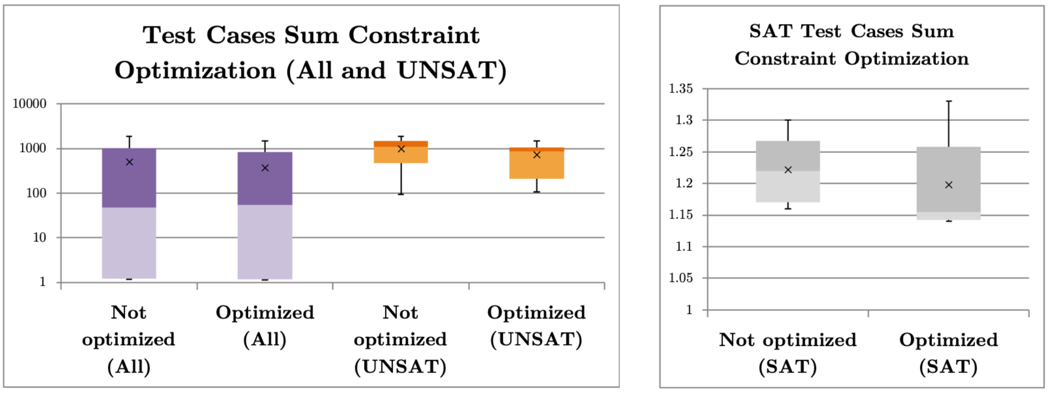


Figure 2 Execution time distributions of the test cases for the Geq/Leq optimization on a logarithmic scale [in seconds].



(a) Execution time distribution on a logarithmic scale for all and the `UNSAT` test cases.

(b) Execution time distribution on a linear scale for all and the `SAT` test cases.

Figure 3 Distributions of runtime for the sum constraint optimization experiment.

the `UNSAT` case as shown in Figure 3a. However, in 45% of the examples, the original version outperformed the optimized encoding, and in the case of `SAT` the execution times remain almost constant between the original and optimized version (see Figure 3b). Especially if the result is `UNSAT`, searching for a solution can still take a long time despite optimizations.

4.3.3 Table Constraints

Overall, 28 test cases from TS5 were used within this evaluation. The maximal execution time was limited to 90 seconds for the MINION solver and to 3 minutes for Z3. This means that the corresponding test case failed if no equivalent solution could be found within this time range. This was the case for five test cases using the SMT-LIB2 translation of the original compilation procedure.

The experiments showed that the original version required 47.58 (*Median*=15.24, *Standard Deviation*=65.71, *Min*=0.11, *Max*=180.00) seconds on average while the optimization reduces this number to 0.45 (*Median*= 0.33, *Standard Deviation*= 0.37, *Min*= 0.06, *Max*= 1.83) seconds. Figure 4 presents the execution time distribution for the initial solution and the optimization. As the figure indicates, the optimization has led to a notable improvement in performance on all cases. Tests that initially timed out and all other tests (`SAT` and `UNSAT`) are usually executed in less than one second after the optimization, making this the optimization with the greatest performance gain.

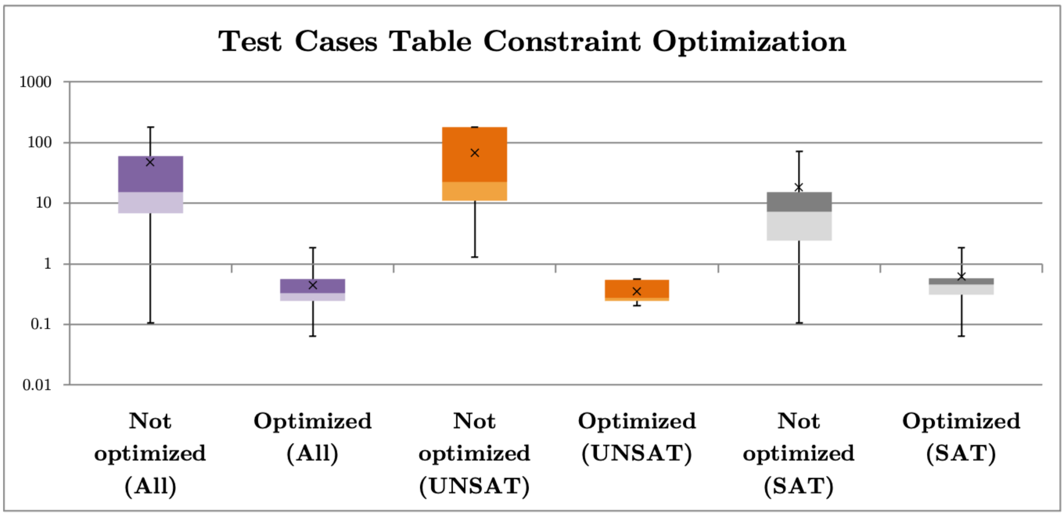


Figure 4 Distributions of the test cases for the `table` constraint optimization on a logarithmic scale.

5 Conclusions

In this paper, we have presented a comprehensive compilation approach for translating MINION models into their corresponding SMT-LIB2 encoding. To improve the execution time of the generated SMT-LIB2 constraints, we implemented several optimizations and assessed them in an empirical evaluation. From the experiment data, we could conclude that the Geq/Leq optimization did have a positive effect on the performance in comparison to the original encoding, in particular in the `UNSAT` case. For the Sum-constraints optimization, we could not generalize such a finding. While there are instances where the optimization exhibits noticeable performance enhancements, this method did not necessarily provide an improvement in all test cases. The Table-constraints optimization was the most impactful, as it drastically reduced the execution time across all scenarios. The Z3 solver generally shows better performance in handling complex and `UNSAT` instances compared to MINION, especially after optimizations. However, for simpler and smaller problems, MINION can still be very effective and sometimes faster

While certain MINION constraints remain unsupported due to unique algorithms or CSP-specific issues, our methodology provides a solid foundation for future enhancements and extensions. Overall, this compilation methodology not only facilitates seamless translation but also contributes to improved efficiency and reliability in solving constraint satisfaction problems. The next logical step for future work is to evaluate the generated SMT-LIB2 models on other solvers such as Yices [15] or CVC5 [7].

References

1 CSPLib Language Essence. http://www.csplib.org/Languages/Essence.

2 CSPLib Language EssencePrime. http://www.csplib.org/Languages/EssencePrime.

3 Model Evaluation - Supplemental material. https://github.com/bhoferTU/Abstraction-Levels-for-Model-based-Software-Debugging.

4 Rui Abreu, Birgit Hofer, Alexandre Perez, and Franz Wotawa. Using Constraints to Diagnose Faulty Spreadsheets. *Software Quality Journal*, 23:297–322, 2015.

5 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

6 Simon Ausserlechner, Sandra Fruhmann, Wolfgang Wieser, Birgit Hofer, Raphael Spork, Clemens Mühlbacher, and Franz Wotawa. The Right Choice Matters! SMT Solving Substantially Improves Model-Based Debugging of Spreadsheets. In *2013 13th International Conference on Quality Software, Najing, China, July 29-30, 2013*, pages 139–148. IEEE, 2013. doi:10.1109/QSIC.2013.46.

7 Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.

8 Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

9 Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret. Solving constraint satisfaction problems with SAT modulo theories. *Constraints*, 17:273–303, 2012.

10 David R Cok et al. The SMT-LIBv2 language and tools: A tutorial. *Language c*, pages 2010–2011, 2011.

11 Francesco Contaldo, Patrick Trentin, and Roberto Sebastiani. From MINIZINC to optimization modulo theories, and back. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21–24, 2020, Proceedings 17*, pages 148–166. Springer, 2020.

12 Ewan Davidson, Özgür Akgün, Joan Espasa, and Peter Nightingale. Effective encodings of constraint programming models to SMT. In *Principles and Practice of Constraint Programming: 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7–11, 2020, Proceedings 26*, pages 143–159. Springer, 2020.

13 Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

14 Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

15 Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

16 Stephan Frühwirt. A Practical Approach to the Compilation of MINION Constraint Models into SMT-LIB2 Models. Master's thesis, Graz University of Technology, 2023.

17 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*, page 98–102, NLD, 2006. IOS Press.

18 C. Gomes, H. Kautz, A. Sabharwal, and B. Selman. *Satisfiability Solvers*. Number 3 in Foundations of Artificial Intelligence. Elsevier, 2008.

19 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*, pages 263–268. IEEE, 2014.

20 Christopher Jefferson, Lars Kotthoff, Neil Moore, Peter Nightingale, Karen E Petrie, and Andrea Rendl. The Minion Manual – Minion Version 0.9. 2009.

21 Thomas W Kelsey, Lars Kotthoff, Christopher A Jefferson, Stephen A Linton, Ian Miguel, Peter Nightingale, and Ian P Gent. Qualitative modelling via constraint programming. *Constraints*, 19:163–173, 2014.

22 David G. Mitchell. On Correctness of Models and Reformulations (Preliminary Version). In *PTHG 2020: The Fourth Workshop on Progress Towards the Holy Grail*, 2020. URL: `https://freuder.files.wordpress.com/2020/09/revised-on_correctness_of_models_and_reformulations.pdf`.

23 Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., 2 edition, 2004.

24 Iulia Nica, Franz Wotawa, Roland Ochenbauer, Christian Schober, Harald Hofbauer, and Sanja Boltek. Model-based simulation and configuration of mobile phone networks – the SIMOA approach. In *Proceedings of the ECAI 2012 Workshop on Artificial Intelligence for Telecommunications and Sensor Networks*, pages 12–17, 2012.

25 Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

26 Charles Prud'homme and Jean-Guillaume Fages. Choco-solver: A Java library for constraint programming. *Journal of Open Source Software*, 7(78):4708, 2022. `doi:10.21105/joss.04708`.

27 Microsoft Research. Z3 - the theorem prover, may 2023. Git Repository. URL: `https://github.com/z3prover/z3`.

28 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*, pages 325–330. IEEE, 2010.

29 Franz Wotawa, Mihai Nica, and Iulia Moraru. Automated debugging based on a constraint model of the program and a test case. *J. Log. Algebraic Methods Program.*, 81(4):390–407, 2012. URL: `https://doi.org/10.1016/j.jlap.2012.03.002`, `doi:10.1016/J.JLAP.2012.03.002`.

A Compilation methodology for all constraints

<table>
<tr><th>MINION constraint</th><th>SMT-LIB2 Conversion</th></tr>
<tr><td>

`abs(x,y)` ensures that $x = |y|$. This means that x is the absolute value of y.

</td><td>

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

</td></tr>
<tr><td>

`alldiff(A)` ensures that each element of A takes a different value.

</td><td>

```
(assert
  (forall ((i Int) (j Int))
    (=>
      (and
        (< 0 i n)
        (< 0 j n)
        (= (select A i) (select A j))
      )
      (= i j)
    )
  )
)
```

</td></tr>
<tr><td>

`alldiffmatrix(A, c)` ensures that in each row of the multidimensional vector A the constant value c appears exactly once.

</td><td>

```
(define-fun-rec fun_alldiffmatrix
((i Int) (j Int) (acc Int)) Bool
  (ite (>= j m)
    (= acc 1)
    (ite (= (select A i j) c)
      (ite (= acc 1)
        false
        (fun_alldiffmatrix i (+ j 1) 1)
      )
      (fun_alldiffmatrix i (+ j 1) acc)
    )
  )
)
(assert (and
  (fun_alldiffmatrix 0 0 0)
    ...
  (fun_alldiffmatrix n 0 0)
  )
)
```

</td></tr>
<tr><td>

`difference(x, y, z)` ensures that $z = |x - y|$.

</td><td>

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

</td></tr>
</table>

`diseq(x, y)` ensures that the two variables have a different value.

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

`div(x, y, z)` ensures that $\lfloor \frac{x}{y} \rfloor = z$ and is always `false` in case of $y = 0$. The MINION implementation of the division differs from the standard implementation of the division in SMT-LIB2. Jefferson et al. [20] presents the following examples: $\frac{10}{3} = 3$, $\frac{-10}{3} = -4$, $\frac{10}{-3} = -4$ and $\frac{-10}{-3} = 3$.

```
(assert
  (and
    (distinct y 0)
    (ite
      (or
        (and (> x 0) (> y 0))
        (and (< x 0) (> y 0))
      )
      (= z (div x y))
      (=
        (div (- x) (- y))
        z
      )
    )
  )
)
```

`element(A,i,e)` ensures that $A[i] = e$, where $0 \leq i \leq n$. The constraint is `false`, if i is outside the index range.

```
(assert
  (and
    (= (select A i) e)
    (< i n)
  )
)
```

`element_one(A, i, e)` is identical to `element(A,i,e)`. However, A is indexed from 1.

```
(assert
  (and
    (= (select A (- i 1)) e)
    (< (- i 1) n)
  )
)
```

`eq(x, y)` ensures that two variables take equal values. 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.

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

```
BOOL b1
BOOL b2
eq(b1, 0)
eq(1, b2)
```

therefore translates to

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

`gacalldiff(A)` is identical to `alldiff` but this constraint enforeces Generalized Arc Consistency (GAC) in MINION.

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

```
(assert
  (>=
    (+
      (ite
        (distinct
          (select A 0) (select B 0)
        )
        1 0
      )
      ...
      (ite
        (distinct
          (select A n) (select B n)
        )
        1 0
      )
    )
    c
  )
)
```

`ineq(x, y, c)` ensures that $x \leq y + c$. This constraint can be used to express $x \leq y$ iff $c = -1$.

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

`lexleq(A, B)` ensures that A is lexicographically less than or equal to B, where A and B are both of same length.

```
(define-fun-rec fun_lexleq ((i Int))
Bool
  (ite
    (>= i n)
    true
    (ite
      (> (select A i) (select B i))
      false
      (ite
        (< (select A i) (select B i))
        true
        (fun_lexleq (+ i 1))
      )
    )
  )
)
(assert (fun_lexleq 0))
```

`lexless(A, B)` ensures that A is lexicographically less than B, where A and B are both of same length.

```
(define-fun-rec fun_lexless ((i Int))
   Bool
  (ite
    (>= i n)
    false
    (ite
      (> (select A i) (select B i))
      false
      (ite
        (< (select A i) (select B i))
        true
        (fun_lexless (+ i 1))
      )
    )
  )
)
(assert (fun_lexless 0))
```

`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 $\sum_i A[i] \neq B[i] \geq c$.

```
(assert
  (>=
    (+
      (ite (= (select A 0) v1) 1 0)
      ...
      (ite (= (select A n) vn) 1 0)
    )
    c
  )
)
```

`max(A, x)` ensures that x is equal to the maximum of any element of A.

```
(assert
  (and
    (or
      (= x (select A 0))
      ...
      (= x (select A n))
    )
    (>= x (select A 0))
    ...
    (>= x (select A n))
  )
)
```

`min(A, x)` ensures that x is equal to the minimum of any element of A.

```
(assert
  (and
    (or
      (= x (select A 0))
      ...
      (= x (select A n))
    )
    (<= x (select A 0))
    ...
    (<= x (select A n))
  )
)
```

`minuseq(x, y)` ensures that $x = -y$.

```
(assert (= x (- y)))
```

`modulo(x, y, z)` ensures that $x \bmod y = z$. The constraint is always `false` when $y = 0$. The MINION implementation of the modulo operator differs from the standard implementation of the modulo operator in `SMT-LIB2`. Jefferson et al. [20] presents the following examples: $3 \bmod 5 = 3$, $-3 \bmod 5 = 2$, $3 \bmod -5 = -2$ and $-3 \bmod -5 = -3$.

```
(assert
  (and
    (distinct y 0)
    (ite
      (or
        (and (> x 0) (> y 0))
        (and (< x 0) (> y 0))
      )
      (= (mod x y) z)
      (ite
        (and (> x 0) (< y 0))
        (=
          (mod (- x) (- y))
          (- z)
        )
        (=
          (mod (- x) y)
          (- z)
        )
      )
    )
  )
)
```

`mod_undefzero(x, y, z)` is identical to `modulo`, except the constraint is always `true` when $y = 0$.

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

```
(assert
  (and
    (or
      (= t00 (select A 0))
      (= t01 (select A 1))
      (= t02 (select A 2))
    )
    (or
      (= t10 (select A 0))
      (= t11 (select A 1))
      (= t12 (select A 2))
    )
    (or
      (= t20 (select A 0))
      (= t21 (select A 1))
      (= t22 (select A 2))
    )
  )
)
```

`occurrence(A, c, x)` ensures that the value x occurs exactly c times in A.	<pre>(assert
	  (=
	    c
	    (+
	      (ite (= (select A 0) x) 1 0)
	      ...
	      (ite (= (select A n) x) 1 0)
	    )
	  )
	)</pre>

`occurrencegeq(A, c, x)` ensures that the value x occurs at least c times in A.	<pre>(assert
	  (>=
	    c
	    (+
	      (ite (= (select A 0) x) 1 0)
	      ...
	      (ite (= (select A n) x) 1 0)
	    )
	  )
	)</pre>

`occurrenceleq(A, c, x)` ensures that the value x occurs at most c times in A.	<pre>(assert
	  (<=
	    c
	    (+
	      (ite (= (select A 0) x) 1 0)
	      ...
	      (ite (= (select A n) x) 1 0)
	    )
	  )
	)</pre>

`pow(x, y, z)` ensures that $x^y = z$.	<pre>(assert
	  (ite
	    (= y 0)
	    (= z 1)
	    (= z (^ x y))
	  )
	)</pre>

`product(x, y, z)` ensures that $x \times y = z$. For Boolean variables, the product constraint acts as conjunction.

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

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

`reify(Constraint, b)` ensures that the constraint must be satisfied if $b =$ `true` and the constraint must not be satisfied if $b =$ `false`.

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

`reifyimply(Constraint, b)` ensures, an implication between b and the constraint which means $b \implies$ Constraint.

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

`sumgeq(A, c)` ensures that $\sum_i A_i \geq c$.

```
(assert
  (>=
    (+
      (select A 0)
      ...
      (select A n)
    )
    c
  )
)
```

`sumleq(A, c)` ensures that $\sum_i A_i \leq c$.

```
(assert
  (<=
    (+
      (select A 0)
      ...
      (select A n)
    )
    c
  )
)
```

`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.

```
(assert
  (or
    (and
      (= t00 (select A 0))
      (= t01 (select A 1))
      (= t02 (select A 2))
    )
    (and
      (= t10 (select A 0))
      (= t11 (select A 1))
      (= t12 (select A 2))
    )
    (and
      (= t20 (select A 0))
      (= t21 (select A 1))
      (= t22 (select A 2))
    )
  )
)
```

`w-inintervalset(a, [c1, c2, c4, c5])` ensures that $c1 \leq x \leq c2, c3 \leq x \leq c4 \ldots$ holds. The interval list must be given in numerical (strictly monotonously rising) order.

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

`w-inrange(x, [c1, c2])` ensures that $c1 \leq x \leq c2$.

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

`w-inset(x, A)` ensures that x equals one of the values in the given set.

```
(assert
  (or
    (= x v1)
    ...
    (= x vn)
  )
)
```

`w-literal(x, c)` ensures that $x = c$.

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

`w-notinrange(x, [c1, c2])` ensures that $x < c1$ or $x > c2$.

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

`w-notinset(x, [c1, c2, ..., cn])` ensures that x is not equal to any of the values in the given set.

```
(assert
  (and
    (distinct x c1)
    (distinct x c2)
    ...
    (distinct x cn)
  )
)
```

`w-notliteral(x, c)` ensures that $x \neq c$.

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

`watched-and(Constraint1, Constraint2, ..., Constraintn)` ensures that all constraints are `true`. This constraint may be used in combination with Constraint `reify`.

```
(assert
  (and
    Constraint1
    Constraint2
    ...
    Constraintn
  )
)
```

`watched-or(Constraint1, Constraint2, ..., Constraintn)` ensures that at least one constraint is `true`.

```
(assert
  (or
    Constraint1
    Constraint2
    ...
    Constraintn
  )
)
```

`watchelement(A, i, e)`: see constraint `element`.

`watchelement_one(A, i, e)`: see constraint `element_one`.

`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 $e = 0$.

```
(assert
  (or
    (= (select A i) e)
    (and
      (>= i n)
      (= e 0)
    )
  )
)
```

`watchless(x, y)` ensures that $x < y$.

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

`watchsumgeq(A, c)`: see constraint `sumgeq`.

`watchsumleq(A, c)`: see constraint `sumleq`.

`watchvecneq(A, B)` ensures that A and B are not the same, i.e., $\exists i : A[i] \neq B[i]$.

```
(assert
  (or
    (distinct (select A 0) (select B 0))
    ...
    (distinct (select A n) (select B n))
  )
)
```

`weightedsumgeq([c0, c1, ..., cn], A, x)` ensures that $\sum_i A_i \cdot c_i \geq x$.

```
(assert
  (>=
  (+
    (* c1 (select A 0))
    (* c2 (select A 1))
    ...
    (* cn (select A n))
  x
  )
)
```

`weightedsumleq([c0, c1, ..., cn], A, x)` ensures that $\sum_i A_i \cdot c_i \leq x$.

```
(assert
  (<=
  (+
    (* c1 (select A 0))
    (* c2 (select A 1))
    ...
    (* cn (select A n))
  x
  )
)
```