SIEMENS

Department of Mathematics and Computer Science
Model Driven Software Engineering

A Solver Agnostic Incremental Machine Reasoning Interface

Master Thesis

Bart van Helvert

Supervised By:
Ir. César Santos (Siemens)
Ir. Mauricio Verano Merino (TU/e)
Prof. Dr. Mark van den Brand (TU/e)

External Committee:
Prof. Dr. Hans Zantema (TU/e)

Version 1.1

Eindhoven, July 2021

Abstract

Many applications from formal methods and artificial intelligence use machine reasoning solvers to solve computationally-hard problems. Various types of solvers exist, which all target and optimize for different problem sets. Choosing the correct solver for a problem can therefore be difficult when the problem does not fit into these problem sets or belongs to multiple problem sets. Furthermore, migrating models between solvers is difficult because every solver provides its own unique API. Partial solutions to these problems exist, but they only focus on a single solver type or do not support incremental solving, which is often required in industry applications. In this master thesis we propose a solution to this problem in the form of an embedded C++ interface called MReason. MReason supports SAT, MIP, SMT and CP type solvers and allows for incremental solving.

Contents

Contents iii

1 Introduction 1

2 Solver Aspects 3
2.1 Logic 4
2.2 Arithmetic 5
2.3 Domain-Specific 5
2.4 Incremental Solving 7

3 Related Work 8
3.1 Single Solver Type Interfaces 8
3.2 MiniZinc 8

4 MReason 10
4.1 Front-end 10
4.1.1 Solver Factory 11
4.1.2 AbsSolver 11
4.1.3 AbsSort 12
4.1.4 AbsTerm 12
4.1.5 Op 13
4.1.6 Exceptions 13
4.2 Back-end 13
4.2.1 Abstract Syntax Tree 13
4.2.2 Mixed Integer Programming 14
4.2.3 Satisfiability Modulo Theories 15
4.2.4 Constraint Programming 15

5 Evaluation 16
5.1 Solver Comparison 17
5.2 Check SAT All Comparison 18
5.3 Solver API Comparison 19

6 Future Work 21
6.1 Search Hints 21
6.2 Operator Overloading 21
6.3 Default Constant Sorts 23
6.4 Emulation 23
6.5 Global Constraints 23
6.6 All Solutions Limit 23
6.7 Solver Support 24
6.8 Performance Evaluation 24
6.9 Unified LoggingSolver 24

6.10 Numeric Variable Domains 24

7 Conclusion **25**

Bibliography **26**

Appendix **28**

A Sudoku Model Source Code **29**

Chapter 1

Introduction

Machine reasoning is an area of computer science dedicated to automatically solving constraint satisfaction problems (CSPs). CSPs consist of a set of variables with non-empty domains and a set of constraints on those variables. Solving CSPs is done by finding an assignment for all the variables without violating any of the constraints. CSP is NP-complete, meaning that every problem in NP can be transformed into a CSP in polynomial time. Many problems like: type inference [1], map coloring [2], model checking [3], circuit synthesis [4] and many logic puzzles [5, 6] can be transformed into CSPs.

Several strategies exist for solving CSPs. Boolean satisfiability (SAT), mixed integer programming (MIP), satisfiability modulo theories (SMT) and constraint programming (CP) are all paradigms that focus on solving various forms of CSPs. Theoretically, they are all capable of solving the same set of combinatorial optimization problems. However, in practice, it might be challenging or inefficient to express certain types of problems in certain paradigms.

Today, many different solvers exist for each paradigm. Because there are so many different solvers with different APIs, it is often very difficult to migrate models defined in one solver API to another solver API. Various solutions to this problem exist, however, they often only focus on a single type of solver.

A solution that covers all types of solvers is MiniZinc [7]. MiniZinc is an external domain-specific language (DSL) which supports a wide variety of solvers. The downside of MiniZinc is that it does not support incremental solving, which is often required by industry applications because they often have to solve multiple instances of the same problem with slight variations, which is much more efficient to do with incremental solving. Another downside of MiniZinc is that, because it is implemented as an external DSL, it makes it more difficult to interact with the model from a general purpose programming language.

To solve this issue, we wrote an internal C++ DSL (or interface) that is solver agnostic and supports incremental solving, called MReason. From the C++ interface, bindings to other general purpose programming languages can be created. MReason already has support for Python through the Cython [8] native interface. Interfaces similar to MReason already exist, but they only target a specific solver type. An example of such an interface is SMT-Switch [9]. SMT-Switch is a C++ solver interface for SMT solvers that is inspired by an external DSL for SMT solvers, called SMT-LIB [10]. MReason was build as a fork of SMT-Switch. Contrary to SMT-Switch, the goal of MReason is to not only support SMT solvers, but also SAT, MIP and CP solvers.

This master thesis is commissioned by Siemens Industry Software (SISW) and the original idea for MReason came from SISW. In this master thesis, we show how we built MReason and use MReason to answer the following research questions:

How to build a C++ interface that interacts with SAT, MIP, SMT and CP solvers which supports incremental solving. (RQ1)

Is there any significant overhead when unifying different type of solvers under the same interface? (RQ2)

In this report, first, we give some background information about CSP solving in Chapter 2 and show some related work in Chapter 3. Chapter 4 explains the MReason solver interface from a technical perspective and highlights how different solver types are implemented through the interface. In Chapter 5 we show how we implemented a toy problem in MReason and evaluated its performance. In Chapter 6 we discuss the aspects of MReason that are still not implemented and how they could possibly be implemented. Lastly, we conclude our finding in Chapter 7.

Chapter 2

Solver Aspects

To talk about the MReason interface we first need to introduce the different aspects used to solve CSPs and the various types of solvers involved when solving CSPs. A CSP is defined as a triple $< X, D, C >$, where X is a set of variables, D is a set of non-empty domains on X, and C is a set of constraints on X. There are many types of variables and constraints. Different solvers focus on solving for different types of variables and constraints. In this report we only focus on SAT, MIP, SMT and CP solvers. All these solvers can solve the same set of combinatorial optimization problems but there are major differences in their APIs and in the methodology used to solve problems.

SAT solvers solely focus on solving Boolean variables with constraints in conjunctive normal form (CNF). SAT solvers can solve problems with a large amount of variables, but their API is very restrictive because it only accepts CNF formulas. Where SAT solvers focus in solving in the Boolean domain, MIP solvers are used to solve in the Integer and Real domains. MIP solvers focus on solving numeric variables with numeric and optimization constraints. Both SMT and CP solvers are also able to solve these arithmetic problems, but MIP solvers are usually faster at solving them. SMT is an extension of SAT where additional theories are added to SAT to provide a more expressive interface. These theories usually have their own domain specific solver. Examples of theories are bit vectors, arrays, algebraic data types and uninterpreted functions (empty theory). CP is similar to SMT in terms of its expressiveness but where SMT solvers are usually optimized for formal verification, CP solvers are optimized for problems in the artificial intelligence domain such as scheduling and routing problems.

Table 2.1 shows an overview of the most common solver aspects and which solver types support them. This table is just a general overview of which features are commonly supported.

● = Supported by most solvers; ◐ = Supported by some solvers; - = Not supported;

Solver Type	Logic		Arithmetic				Domain-Specific				
	Propositional	First-order	(Integer) Linear	(Integer) Polynomial	Transcendental	Optimization	Bit Vectors	Algebraic Data Types	Arrays	Empty Theory	Global Constraints
Boolean Satisfiability (SAT)	●[1]	-	-	-	-	-	-	-	-	-	-
Mixed Integer Programming (MIP)	-	-	●	●	-	●	-	-	-	-	-
Satisfiability Modulo Theories (SMT)	●	●	●	●	◐	◐	●	●	●	●	-
Constraint Programming (CP)	●	-	●	●	◐	●	-	●	●[2]	-	●

[1] Conjunctive normal form
[2] Enumeration types

Table 2.1: Solver aspects support for each solver type

2.1 Logic

Logic problems are CSPs with constraints over Boolean domains. Solving logic problems is usually done through a SAT, SMT or CP solver. Logic problems can be categorized by the type of operators that are applied to the Boolean variables. In this section we will discuss both proposition and first-order logic.

Propositional Logic

Propositional logic is a branch of logic and is sometimes also called zeroth-order logic. Propositional logic is able to express statements which can be either true or false and the relations between those statements. Those statements are called propositions. Propositions can be related to other propositions to create new propositions by using logical operators like `and`, `or` and `if`. An example of a propositional formula is $p \vee q$, where p and q are propositions and $\vee$ is the operator representing a disjunction of p and q. Propositional logic is supported by most solvers. Which operators are supported can however vary per solver. This is not a problem because most operators can be expressed by combining other operators. A set of operators that is able to express any truth table is called functional complete. Most solvers, however, provide support for a much wider range of operators to make it easier to model problems.

An important sub-class of proposition logic, which is used by SAT solvers is CNF. A constraint in CNF is a conjunction of clauses, where clauses are a disjunction of literals and a literal is either a Boolean variable or the negation of a Boolean variable. An example of a CNF constraint is $(p \vee q) \wedge (\neg r \vee s)$. In practice, CNF constraints are usually not hand written but generated by an encoder such as PBLib [11].

First-order Logic

First-order logic is an extension of propositional logic that adds quantification over variables and predicates. Predicates are Boolean functions denoted as $P(x)$, where P is a Boolean function on variable x. Existential and universal quantification can be used to quantify over sets of variables. $\forall_x P(x)$ for example represents that for all x, $P(x)$ must hold. Some solvers support predicate logic and have support for existential quantification and universal quantification. Just like in propositional logic supporting either the `for all` ($\forall$) or `exists` ($\exists$) quantifier makes the interface functionally complete. First-order logic only specifies quantification over variables, higher-order logic also allows quantification over nested sets. Second-order logic, for example, allows quantification over sets and third-order logic allows quantification over sets of sets. Solving higher-order

logic is not supported by most solvers, but a handful of solvers [12] exist that can solve these types of problems.

2.2 Arithmetic

Arithmetic problems are CSPs with constraints over numeric domains. Solving arithmetic problems can either be finding a satisfying assignment or optimizing an objective function. In arithmetic optimization problems, the objective function is either a minimization or maximization function. An example of an arithmetic optimization problem is:

$$\begin{aligned} \text{max} \quad & 4x + 2y && (2.1) \\ \text{subject to} \quad & x + y \leq 10 && (2.2) \\ & 2x + 3y \leq 23 && (2.3) \\ & x, y \in \mathbb{Z} && (2.4) \end{aligned}$$

Here, $\max(4x + 2y)$ is the objective function and x and y are decision variables. Arithmetic problems are solved by finding an assignment of decision variables to satisfy the objective function without violating the constraints. In the example above, an assignment for x and y should be found that maximizes $4x + 2y$ without $x + y \leq 10$ and $2x + 3y \leq 23$ becoming false.

Mathematical Programming

Solving arithmetic problems is called mathematical programming (MP). There are many subclasses of MP problems. MP problems are classified by three different aspects: the type of numeric domain, the type of the objective function and the type of constraints.

The first classification is the type of the numeric domain. When no domain type is mentioned it is assumed that all decision variables are real numbers. Another popular domain type in constraint solving is the integer domain. In integer programming (IP), all decision variables are integers. When the variables in a problem are a mix between real variables and integer variables this is called mixed integer programming (MIP).

The second classification for MP problems is by the type of objective function. In Linear programming (LP) for example, the objective function is a linear polynomial. Besides linear programming, other classes of polynomial programming exist, such as quadratic programming (QP). Non-polynomial objective functions are called transcendental objective functions and solving them is called transcendental programming. Examples of transcendental functions are logarithmic, exponential and trigonometric functions.

Lastly, MP problems can be classified by their constraint type. If no constraint type is specified it is assumed that the constraints are linear. When the constraints are of the same type as the objective function the `constrained` term is added to the definition. E.g. QP with quadratic constraints is called quadratically constrained programming (QCP). When specifying a polynomial constraint problem the type represents an upper bound on the degree of the constraints. QCP for example, also allows linear constraints.

The complexity of these aspects influences how easy it is to solve a problem. LP problems are the easiest to solve. Solving non-linear problems, however, is more difficult. Adding integer constraints to the variables also makes the problems more difficult to solve. Non-linear mixed integer programming has been proven to be undecidable [13]. Modern solvers can, however, still deal with these types of problems by making the integer domain finite.

2.3 Domain-Specific

Some aspects of machine reasoning are specific to a single domain or type of solver. Throughout the years however, solvers have adapted aspects from each other. Aspects that were previously

only supported by a single type of solver are now also embedded in other types of solvers to a limited degree. Examples of these domain-specific aspects are bit vectors, arrays, algebraic date types, uninterpreted functions and global constraints.

Bit Vectors

A bit vector is a sequence of bits with a fixed size. Bit vectors can be converted to Boolean logic through a process called bit blasting. This allows running bit vector problems on SAT solvers. The advantage of using bit vectors over Boolean logic is that bit vectors can be used to represent numbers and perform arithmetic operations.

Arrays

Arrays are fixed size sequences of data of the same type. Interacting with arrays in SMT and CP solvers is done in a different way. SMT arrays utilize the select-store axioms [14] and provide two operations on arrays; 'select' and 'store'. Select takes a value from the array at a specified index, while store returns a modified copy of the array where it puts the specified value at a specified index. In CP solvers, arrays do not use the select-store axioms. Instead, they allow specifying constraints by indexing the declared array, similar to most general purpose programming language.

Algebraic Data Types

Algebraic data types are composite types that can be created by combining existing types. There are two types of algebraic data types: product types and sum types. Product types contain a set of fields. Each field has its own type and can be assigned a value. Examples of product types are tuples and records. Sum types, on the other hand, can only be assigned a single value. Sum types are defined as a set of types, called variants, that can each have a constructor and parameters. When assigning a sum type, the value should be of one of the types variants. Sum types can also be defined recursively. An example of a recursive sum type is `Tree = Node(Tree, Tree) | Leaf`. Enumeration types are a special case of sum types where all variants have zero parameters.

Empty Theory

Empty theory, or sometimes also called uninterpreted functions are functions without an implementation. Uninterpreted functions are satisfiable for any possible interpretation, given it satisfies all other constraints. Uninterpreted functions without arguments are sometimes also called uninterpreted constants.

Global Constraints

Global constraints are commonly used constraints, which are directly supported by the solver API. Global constraints are solved using a constraint specific solving algorithm, making them very efficient. Many different type of global constraints exist. For instance, the `alldifferent` constraint adds the constraint that all of its arguments should have a different assignment. Instead of doing pairwise inequality, the solver can utilize a bipartite matching method [15] which can efficiently solve the `alldifferent` constraint.

A special class of global constraints are soft global constraints. The purpose of soft global constraints is to find an alternative solution when no assignment can be found that satisfies all constraints. The `soft-alldifferent` constraint; for example, can be used to minimize the amount of equal variable assignments between its arguments.

2.4 Incremental Solving

Incremental solving is a technique to increase the solving performance when doing multiple solve calls in a row on a shared set of variables. The solver does this by remembering assumptions made in solve calls for future solve calls. Take for example, the constraint $p > q \land x < y$ which after solving, has the following assignment: $p = 2$, $q = 1$, $x = 3$ and $x = 4$. Then, in a subsequent solve the constraint $p > q \land x \geq y$ has to be solved. Using incremental solving the solver can already know that $p = 2$ and $q = 1$ and it only has to find a satisfying assignment for $x \geq y$.

When incremental solving is supported by a solver, it usually requires no extra effort from the user. Some solvers, however, provide APIs to modify the model between solve calls. An example of this is the push/pop mechanism that most SMT solvers provide. The push/pop mechanism allows the user to maintain a stack of assertions and lemmas. When calling `push` all assertions after that are pushed onto a stack. When calling `pop` all assertions pushed on the stack after the latest `push` call are popped from the stack and all lemmas that where found during solving are removed from the model.

Another tool to aid incremental solving is local solve assumptions. These assumptions can be passed into the solve call and they will only hold for that particular solve call. Most SMT solvers support local solve assumptions.

It is also possible to modify the model through half-reification. This can be done by introducing a new variable and adding an implication from the newly introduced variable to the assumption. For example, when asserting the formula $q \lor r$ it would half-reified to $p \Rightarrow (q \lor r)$. Then, in a later solve it is possible to remove this assumption by assuming $\neg p$. The advantage of this approach over the other approaches is that it does not require the assumptions to be garbage collected by the solver. The disadvantage, however is that after disabling the constraint, it is not possible to enable it in a later solve call. Another disadvantage of this approach is that it introduces a lot of extra variables that can slow down the solving process.

Chapter 3

Related Work

Many common interfaces for solvers already exist. All of them, except for MiniZinc only support a single solver type. In this chapter we will briefly mention all the most popular existing interface and how they work.

3.1 Single Solver Type Interfaces

For SAT solvers the most popular interface is DIMACS [16]. DIMACS can be used to write problems in various sub-classes of proposition logic. One of these sub-classes is CNF, which makes it a very suitable format for SAT solvers. Problems in DIMACS are usually not hand written, but computationally generated because the format is difficult to read for humans and SAT problems are usually very large. Another popular interface for interacting with SAT solvers is IPASIR. IPASIR is a C interface that allows interacting with many different SAT solvers APIs through a single interface.

For MIP solvers there are several external DSLs available, most of these DSLs are very similar with the main difference being that some DSLs are meant as exchange formats while other DSLs are made for human problem writing. One of the most popular DSLs is AMPL [17], which is a more human friendly format. Besides external DSLs, OR-Tools [18] provides a C++ API with bindings to Python and Java for solving MIP problems.

SMT-LIB (or SMT-LIB2) is the most widely used DSL for solving SMT problems. Almost every SMT solver supports SMT-LIB. Because it is so widely supported, most solver APIs are inspired by SMT-LIB. SMT-Switch, a C++ SMT interface on which MReason is based, is also inspired by SMT-LIB. Besides SMT-Switch, metaSMT [19] also provides a C++ interface for solving SMT problems. Compared to SMT-Switch, metaSMT only supports a select few solver aspects and uses C++ templates to directly convert metaSMT API calls to solver API calls whenever possible. This reduces the amount of overhead caused by the interface but it also makes it more difficult to implement new solvers. Furthermore, meta-programming concepts like C++ templates are usually not supported in other general purpose programming languages, making it more difficult to support these languages through the same interface.

A shared interface for CP solvers is XCSP3 [20]. XCSP3 is a language which is embedded in the Extensible Markup Language (XML) [21]. XCSP3 supports a wide variety of solvers.

3.2 MiniZinc

MiniZinc is a DSL for constraint modeling. MiniZinc supports a wide variety of SAT, MIP, SMT and CP solvers. Besides the MiniZinc constraint modeling language, Minizinc also provides an integrated developer environment (IDE) for the MiniZinc language and an intermediate file format called FlatZinc.

The solvers supported by MiniZinc never directly interact with the model written by the user. MiniZinc first flattens the model to a simplified version of the MiniZinc language called FlatZinc. After converting a model to its FlatZinc representation, the model is passed to the solver. This can either be done directly in memory or by storing it in a temporary file, the approach depends on the solver. The reason for using FlatZinc is that FlatZinc is easier to implement for solvers compared to the full MiniZinc modeling language. This allows MiniZinc to be very feature rich and expressive for its users while providing a simple shared interface for solver developers. Because of its design as an external DSL, MiniZinc makes it difficult to incrementally modify the model from a general purpose programming language. Minizinc, therefore also does not support incremental solving.

Chapter 4

MReason

The MReason design is split into two parts; the front-end and the back-end. The front-end is the part which the user interacts with and is defined in terms of C++ classes with pure virtual methods, also called interfaces. The back-end part of the interface are the implementations of these interfaces. The back-end part needs to be implemented on a per solver API basis and is invisible to the user. An example of the frond-end/back-end relation is shown in Figure 4.1. The

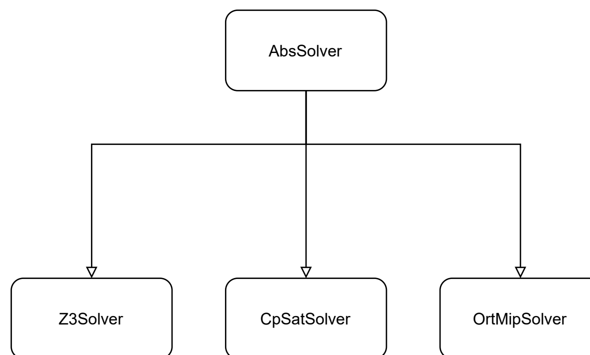

Figure 4.1: MReason front-end/back-end architecture

diagram shows the front-end interface `AbsSolver` and its back-end implementations `Z3Solver`, `CpSATSolver` and `OrtMipSolver` which all implement the `AbsSolver` interface. In this architecture the user never directly interacts with the solver implementations. This makes it so solver implementations can be added and modified without any breaking changes in the interface. The source code for MReason can be found at https://gitlab.tue.nl/mReason/mreason.

4.1 Front-end

The interface front-end defines how user interacts with MReason. The interface is implemented in C++ with bindings to Python. In this chapter we will focus on the C++ interface because the Python interface operates by calling the C++ interface. Other languages can be implemented in a similar way. One of the goals when developing the front-end part of the interface is that it is stable and solver agnostic. This is important because we do not want to have any breaking changes in the interface when adding support for new solvers.

The interface front-end consists of three main classes; `AbsSolver`, `AbsTerm` and `AbsSort`. Adding a solver back-end can be done by providing solver specific implementations for each of those interfaces. When interacting with the interface, the user interacts with shared pointers to the aforementioned abstract classes. MReason also provides three different type aliases; `Solver`, `Term` and `Sort` that represent shared pointers to the previously mentioned interfaces respectively.

These shared pointers serve as a garbage collector by using a reference counting algorithm. This makes it so that the user does not have to worry about manually freeing up the `AbsSolver`, `AbsTerm` and `AbsSort` objects. A diagram of this design is shown in Figure 4.2.

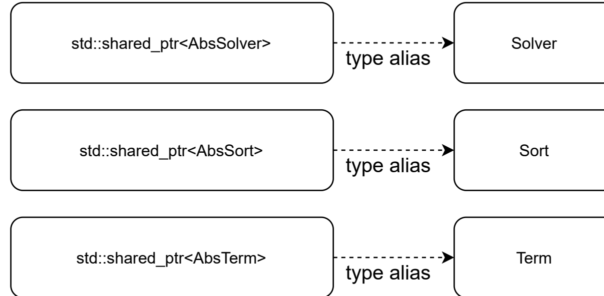

Figure 4.2: Shared pointer design

4.1.1 Solver Factory

The first step when creating a MReason model is creating a `Solver` object. This can be done using a solver factory. Solver factories are classes with a single static method which can be used to create a new `Solver` object. Every solver needs its own solver factory. Creating a solver can be done by calling `SolverFactory#create` which returns a `Solver` object. It is not recommended to call the Solver constructor directly because some solvers might require additional configuration, which is done in the solver factory method. When multiple solvers share the same interface back-end they still have their own solver factory. An example of this is shown in Figure 4.3.

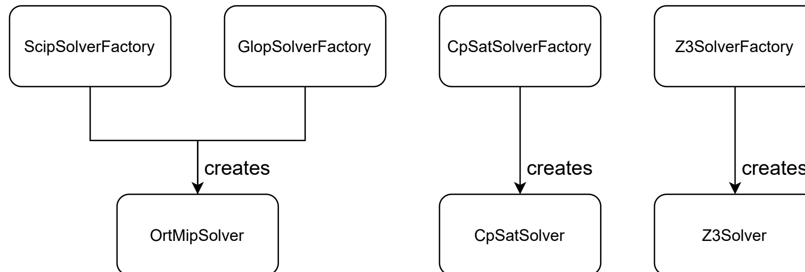

Figure 4.3: Solver factory design

4.1.2 AbsSolver

`AbsSolver` represents a solver instance of a specific solver. The `AbsSolver` is the entry point to create constraint models and allows for interacting with the underlying solver through a common interface. The user can interact with the `AbsSolver` interface through the `Solver` type alias like shown in Figure 4.2.

Creating a model starts with creating a `Sort`. This is done through the `AbsSolver#make_sort` method. Sorts are like types for terms. Sorts are created by passing in a `SortKind`. `SortKind` is an enumeration type that represents the kind of Sort to create. MReason supports eight different kinds of Sorts: `Bool`, `Int`, `Real`, `Array`, `BitVector`, `Function`, `Uninterpreted` and `DataType`. Which `SortKind`s are supported depends on the underlying solver. Uninterpreted functions and constants, for example, are usually not supported by SAT, MIP and CP solvers. Table 2.1 shows a generalized overview of supported `SortKinds`, based on solver type.

From a `Sort` it is possible to create a `Term`. Terms are used to model constraints and objectives. A term can either be a symbol (decision variable), a value (constant) or an expression built from

symbols and values using operators. To create a symbol the `AbsSolver#make_symbol` method has to be used, it takes in a symbol name and a `Sort`. Creating value and expression terms is done by using the `AbsSolver#make_term` method. Value terms are created from C++ primitives which can either be a `bool`, `int` or `double` type which represent the `Boolean`, `Int` and `Real` `SortKind` respectively. Expression terms can be made by combining existing terms with an `Op`. When creating an expression term with more than two terms, MReason follows the same associativity rules as SMT-LIB. To increase performance, some left-associative operators are, however, encoded as polyadic expressions in the interface.

Creating terms will not automatically add them to the solver and have them be considered as a constraint or objective while solving. To do this, the term needs to be passed into either `AbsSolver#assert_formula` or `AbsSolver#add_objective` to add a constraint or objective respectively. To add a constraint, the term must be characterized by a `Boolean` `SortKind`. When adding an objective the term must be an expression term with either the `Max` or `Min` operator. When these conditions are not met, MReason will throw an `IncorrectUsageException` at runtime.

Solving the asserted constraints can be done by calling `AbsSolver#check_sat`, which will return a result enumeration type which can either be `SAT`, `UNSAT` or `UNKNOWN`. Finding all solutions to a problem can be done by calling `check_sat` and adding the blocking clause to the model. The blocking clause is a negation of the found solution. For example, lets assume a problem with variables p and q that get assigned 1 and 2 respectively by the solver. Then a blocking clause can be added in the form of $(p \neq 1) \vee (q \neq 2)$ which will remove $(p = 1) \wedge (q = 2)$ from the possible solution set in the next solve call. This approach, however can be very slow for solvers that do not support incremental solving. The naive method of adding blocking clauses can be improved on [22] by adding minimal blocking clauses. This method, however, still requires solving the same model multiple times for solvers that do not support incremental solving. Most of these solvers have built in methods to find all solutions which can give massive performance increases over manually adding blocking clauses. To facilitate these optimizations the `AbsSolver` provides the `check_sat_all` method which takes in a callback that gets called on every found solution.

After solving, getting the solution can be done by passing in variables or an objective function to `AbsSolver#get_value`. The `AbsSolver#get_value` method will return a new term which has the result set. On this new term it is possible to call `AbsTerm#get_int` or `AbsTerm#get_double` to retrieve the solution in the form of a C++ primitive `int` or `double` type.

4.1.3 AbsSort

`AbsSort` is an interface that represents the type of a term. An `AbsSort` is backed by a `SortKind` enumeration type that represents basic types like: Booleans, Integers, Reals, bit vectors, functions, uninterpreted functions and data types. The `Sort` type represents a shared pointer to an `AbsSort`. An `AbsSort` should never be created by calling its constructor, but always by calling `AbsSolver#make_sort`. When the `AbsSort` is backed by a bit vector it is possible to get the width of the bit vector by calling `AbsSolver#get_width`. When the `AbsSort` is an uninterpreted function sort, the API provides the `AbsSort#get_uninterpreted_param_sorts` method to retrieve the `Sort`s of the function parameters.

4.1.4 AbsTerm

`AbsTerm` represents a term created by an `AbsSolver`. Terms can either be a symbol, a value or an expression built from symbols and values using operators. Terms can be created by calling `AbsSolver#make_symbol` or `AbsSolver#make_term`. An `AbsTerm` is always backed by an `AbsSort` that represents the type of the `AbsTerm`. The `AbsSort` can be retrieved by calling `AbsTerm#get_sort`.

A term can be iterated over by using a `TermIter`. `AbsTerm#begin` and `AbsTerm#end` both return a `TermIter`, which can be used to iterate over the children of a term or to create a `TermVec`,

a `TermList`, an `UnorderedTermSet` or an `UnorderedTermMap`. To support term iteration, the interface back-end must implement the `TermIterBase` interface.

Retrieving the result of a solve is also done through the `AbsTerm` API by calling `AbsTerm#to_int` or `AbsTerm#to_double` on a value term. This value term can be obtained by calling `AbsSolver#get_value` on a symbol term. This `get_value` call is required because some solvers need the underlying solver object to retrieve the result from the solver.

4.1.5 Op

The `Op` class represents an operator in the interface. Every `Op` is backed by a `PrimOp` enumeration type. MReason supports a wide variety of operators but not all of them are support by every solver. Bit vector operators are, for example, not supported in MIP solvers.

4.1.6 Exceptions

MReason provides a set of custom exceptions to indicate mistakes and errors. The first exception is the `IncorrectUsageException`, which is thrown when the user uses a feature that is not supported by the solver. When a particular feature is implemented but not supported by MReason the `NotImplementedException` is thrown. Lastly, the `InternalSolverException` exception is thrown when an exception is caught from the underlying solver.

4.2 Back-end

The interface back-end is how the interface interacts with the solver APIs. MReason supports seven different interface back-ends and fifteen different solvers, including SMT, CP and MIP solvers. Multiple solvers can be supported under the same back-end when the targeted back-end API supports multiple solvers. Implementing solver support through such an intermediate interfaces makes it very easy to support a wide number of solvers. The downside of this approach, however, is that these intermediate interfaces often do not expose the full solver API which might make it impossible to do certain performance optimizations. In this chapter we will discuss how MReason supports different types of solvers.

4.2.1 Abstract Syntax Tree

The front-end interface separates the creation of terms and the assertion of terms into different calls. This makes it possible to add, remove and reintroduce terms into the model throughout multiple solve calls. Most MIP and CP solvers, however, bundle both steps into a single API call because they do not support incremental solving. To keep the ability to update the model between solve calls, the interface back-end needs to split the creation of the terms and the assertion of the terms for solver APIs that do not do this already. This can be done by maintaining a representation of the model in the form of an abstract syntax tree (AST) at the interface level. When creating a term, the term is stored in this AST without calling the solver back-end. Then, when asserting a term the solver back-end is called and the term is added to the model in the solver.

Another benefit of this lazy approach to forwarding the AST, is that it allows switching between solvers at runtime. Because the whole model is stored at the interface level, the model can be copied over to another solver on demand. This is not possible if the interface forgets about the AST representation after forwarding it to the solver. It is not always possible to retrieve the model back from the solver and even when it is possible, solvers usually change the AST to speed up the solving process. The downside of this approach, is that the model is stored in memory twice which can result in a larger memory consumption.

MReason always stores the AST for MIP and CP solvers because when creating terms in these solver APIs the term is automatically added to the model. For SMT solvers, MReason does not do this but it provides a wrapper class around `AbsSolver` called `LoggingSolver`. The

LoggingSolver stores all the terms created at the interface level and allows for switching between SMT solvers at runtime. This approach allows the user to choose between having the flexibility of the LoggingSolver or the memory efficiency of the regular interface implementations.

4.2.2 Mixed Integer Programming

MReason has support for nine different solvers, namely, CLP [23], CBC [24], GLOP [18], BOP [18], SCIP [25], Gurobi [26], CPLEX [27], Xpress [28], GPLK [29]. All of them are implemented through the OR-Tools[18] MIP interface back-end. The back-end implementation has its own AST implementation, which is shown in Figure 4.4. The figure shows the OR-Tools MIP term data structure, the dotted line represents the variant type and the blue nodes are references to objects stored in the solver. Each term in the OR-Tools back-end is a wrapper around a variant type,

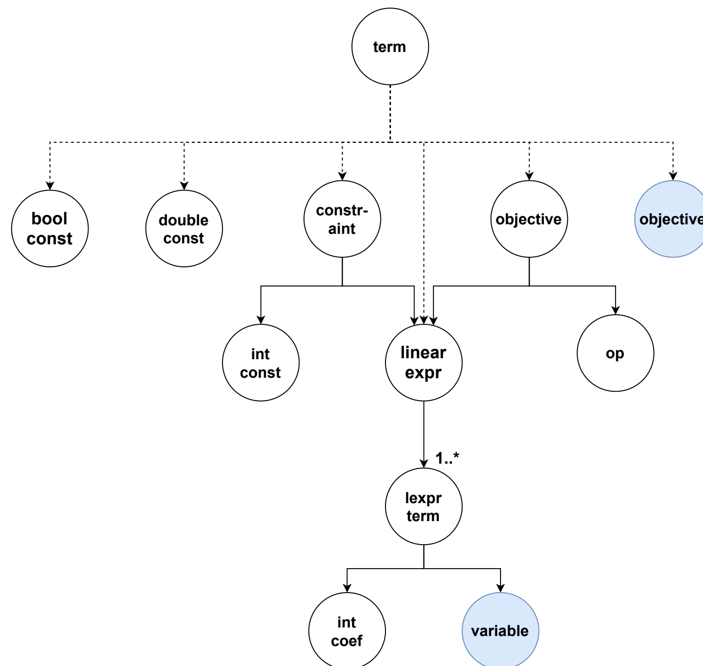

Figure 4.4: OR-Tools MIP AST

which represents either a constant, linear expression, constraint or objective. When the term is a constant, the constant is stored in either a bool or double C++ primitive. When creating a symbol, this symbol is stored as a linear expression. Creating for example the symbol "x" will be encoded as the linear expression $1 * x$. The plus and multiplication operator can be used to build linear expressions which are all stored as a vector of terms. Making a constraint can be done by combining a linear expression with a constant using the $\leq$ operator. Constraints are always stored in a normal form of $a * x \leq b$. The OR-Tools MIP solver back-end, however, also allows using the $<$, $\geq$ and $>$ operators. When using those operators, the constraint is automatically converted to a normal form. Besides constraints, there are objectives, which can either represent a minimization or maximization function. There are two different encodings for objectives; the first encoding represents the objective which is created from the AbsSolver#make_term call when called with either a minimization or maximization operator. When calling AbsSolver#add_objective on this term, the objective is added to the model and a new term is returned. Instead of encoding the objective at the interface level, this new term contains a pointer to the objective in the solver. This pointer is required for retrieving the result of the objective function after a solve call.

4.2.3 Satisfiability Modulo Theories

MReason supports five different SMT solver back-ends, namely, Boolector [30], CVC4 [31], MathSAT5 [32], Yices2 [33], Z3 [34]. All these SMT solvers are implemented by interacting with the native solver APIs directly. Since MReason and most SMT solver APIs are designed to be similar to SMT-LIB it is often trivial to forward the MReason API calls to its corresponding back-end solver APIs. In contrast to MIP and CP solvers, the SMT interface back-end implementations do not need to maintain their own variant of the term AST because the solvers do not automatically add constraints and objectives to the model when they are created. It is however possible to maintain the AST at the interface level by using the `LoggingSolver`. The `LogginSolver` delegates all calls to the underlying `AbsSolver` while storing the AST at the interface level. This can be used to switch between SMT solvers at runtime without redefining the model. In MReason it is only possible to switch between SMT solvers at runtime, switching from and to CP or MIP solvers is currently not supported.

4.2.4 Constraint Programming

MReason has support for one CP solver, namely, CP-SAT [18] which is part of the OR-Tools package. Just like the OR-Tools MIP implementation, the CP-SAT interface back-end implementation has its own AST representation, shown in Figure 4.5. In CP-SAT constants need to be stored as

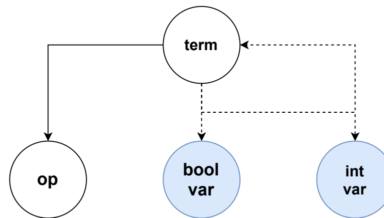

Figure 4.5: CP-SAT AST

variables to be considered for solving, this simplifies the AST because both symbols and constants can be stored as variables in the solver. Terms can also have other terms as their children. These expression terms can be created by combining existing terms with an operator. When creating constants or symbols the operator is set to `None`.

Reified Constraints

The CP-SAT solver interface can only add constraints on variables. It is for example possible to add the constraint $x \wedge y$ but adding the constraint $(x \wedge y) \vee z$ is prohibited because the left hand side of the `or` $(x \wedge y)$ is not a variable. There is, however, one exception to this rule. It is possible to use the implication operator $(\Rightarrow)$ with a constraint on the right hand side. $p \Rightarrow (x \wedge y)$, for example would be a constraint that can be added to a CP-SAT model. We can use this to add constraints like $(x \wedge y) \vee z$ by introducing a new variable p and splitting our original constraint into two separate constraints: $p \Leftrightarrow (x \wedge y)$ and $p \vee z$. Then, we split the first constraint into two separate constraints so that it becomes $p \Rightarrow (x \wedge y)$, $\neg p \Rightarrow (x \wedge y)$ and $p \vee z$. These three constraints are logically equivalent to our original formula. This process is called (full) reification. The CP-SAT interface back-end automatically reifies all constraints, this is done when calling `AbsSolver#assert_Formula` on a term. When reifying a term, all Boolean terms are reified except for the top most term in the AST.

Chapter 5

Evaluation

In this chapter we want to evaluate the interface to check whether we can write a problem and solve it using MReason with different solvers. To do this, we solve a Japanese logic puzzle called Sudoku. There are many variants of the Sudoku puzzle, in this evaluation we will consider the traditional Sudoku puzzle. The traditional Sudoku puzzle is defined as an $N^2 \times N^2$ grid where each cell in the grid can be assigned a number from 1 to N inclusive. The grid is also divided in $N \times N$ sections called regions. Solving the puzzle means finding an assignment for each cell such that all cells in the same row are assigned unique numbers, all cells in the same column are assigned unique numbers, all cells in the same region are assigned unique numbers and every cell is assigned a number between 1 and N^2. When solving a Sudoku, some of the cells are already assigned. With the assigned cells, the solver can deduce the assignment of other cells. The complexity of solving Sudoku depends on the size of the puzzle and the starting assignment. When the Sudoku is big, naturally there are a lot of variables to consider which makes the puzzle more complex. Depending on the starting assignment, it can be difficult to make conclusions about other assignments which also makes the puzzle more complex. A Sudoku problem can have multiple solutions, an empty 9×9 Sudoku has 6.67×10^{21} different solutions, ignoring symmetries like rotation and reflection.

To solve a Sudoku puzzle we define a $N^2 \times N^2$ grid and give each cell an unique identifier like shown in Figure 5.1. After introducing all the variables, we model the constraints. How the

$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$	$x_{0,4}$	$x_{0,5}$	$x_{0,6}$	$x_{0,7}$	$x_{0,8}$
$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$	$x_{1,5}$	$x_{1,6}$	$x_{1,7}$	$x_{1,8}$
$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{2,4}$	$x_{2,5}$	$x_{2,6}$	$x_{2,7}$	$x_{2,8}$
$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	$x_{3,4}$	$x_{3,5}$	$x_{3,6}$	$x_{3,7}$	$x_{3,8}$
$x_{4,0}$	$x_{4,1}$	$x_{4,2}$	$x_{4,3}$	$x_{4,4}$	$x_{4,5}$	$x_{4,6}$	$x_{4,7}$	$x_{4,8}$
$x_{5,0}$	$x_{5,1}$	$x_{5,2}$	$x_{5,3}$	$x_{5,4}$	$x_{5,5}$	$x_{5,6}$	$x_{5,7}$	$x_{5,8}$
$x_{6,0}$	$x_{6,1}$	$x_{6,2}$	$x_{6,3}$	$x_{6,4}$	$x_{6,5}$	$x_{6,6}$	$x_{6,7}$	$x_{6,8}$
$x_{7,0}$	$x_{7,1}$	$x_{7,2}$	$x_{7,3}$	$x_{7,4}$	$x_{7,5}$	$x_{7,6}$	$x_{7,7}$	$x_{7,8}$
$x_{8,0}$	$x_{8,1}$	$x_{8,2}$	$x_{8,3}$	$x_{8,4}$	$x_{8,5}$	$x_{8,6}$	$x_{8,7}$	$x_{8,8}$

Figure 5.1: Sudoku($N = 3$)

problem is split into constraints depends on the user. It is possible to write the whole model into

a single constraint as a conjunction of equations. For clarity, however, we usually split up the model into smaller, easier to understand, self-contained constraints. In the Sudoku example, we decided to add a constraint for each rule in the Sudoku. First, we add a constraint to bound the cells to $[1, N^2]$ in the following constraint:

$$\bigwedge_{i=0}^{(N^2-1)} \bigwedge_{j=0}^{(N^2-1)} (x_{i,j} >= 1 \wedge x_{i,j} <= N^2) \tag{5.1}$$

Then, we add the constraint to make sure that each cell in the same row gets assigned a different value. The constraint looks as follows:

$$\bigwedge_{i=0}^{(N^2-1)} \bigwedge_{j=0}^{(N^2-2)} \bigwedge_{k=j+1}^{(N^2-1)} (x_{i,j} \neq x_{i,k}) \tag{5.2}$$

For each row i the constraint adds a pairwise comparison between the variables in both cells and declares that the assignment cannot be equal. We can add a similar constraint for the columns:

$$\bigwedge_{j=0}^{(N^2-1)} \bigwedge_{i=0}^{(N^2-2)} \bigwedge_{l=i+1}^{(N^2-1)} (x_{i,j} \neq x_{l,j}) \tag{5.3}$$

A two-dimensional variant of this constraint is required for making sure that all assignments in a region are unique:

$$\bigwedge_{i=0}^{(N-1)} \bigwedge_{j=0}^{(N-1)} \bigwedge_{l=i*N}^{(i*N+N-2)} \bigwedge_{k=j*N}^{(j*N+N-2)} \bigwedge_{m=l+1}^{(i*N+N-1)} \bigwedge_{o=k+1}^{(j*N+N-1)} (x_{l,k} \neq x_{m,o}) \tag{5.4}$$

These are all the constraints required to solve the traditional Sudoku. We modeled these constraints in MReason, the source code for this model is shown in Appendix A.

Besides being able to solve the problem, to benchmark our solvers we also need a problem set. Generating Sudoku puzzles can be done by solving for an empty grid. Then after finding a solution we remove assignments to get an initial assignment. In our solution set, after solving we evaluate every assignment and decide whether we want to remove it based on a random generator. Every assignment has a 40% of surviving this evaluation. This makes it so that on average, in our problem set roughly 40% of the cells are assigned but there is still some variation in the amount of already assigned cells.

We try to benchmark as many solvers as possible. MIP solvers do not support these type of problems and the Boolector solver does not support integer variables without encoding them as bit vectors first. This leaves the Yices2, CP-SAT, Z3, CVC4 and MathSAT5 solvers.

5.1 Solver Comparison

In the first benchmark, we generate Sudoku puzzles of different sizes to find out how the interface performs for different problem sizes. For each N value we generate five different Sudoku puzzles and calculate the average solving time. The results are shown in Figure 5.2. The plot shows that CP-SAT significantly outperforms the SMT solvers, this is expected because CP solvers are optimized for artificial intelligence type problems. There also seems to be a large difference between the different SMT solvers. The difference in the performance of these SMT solvers might be explained to how these solvers implement the `all-different` global constraint. Because this constraint is so frequently used, some SMT solvers adopted methods originally used by CP solvers for solving the `all-different` global constraint. Not all SMT solvers, however, have implemented this.

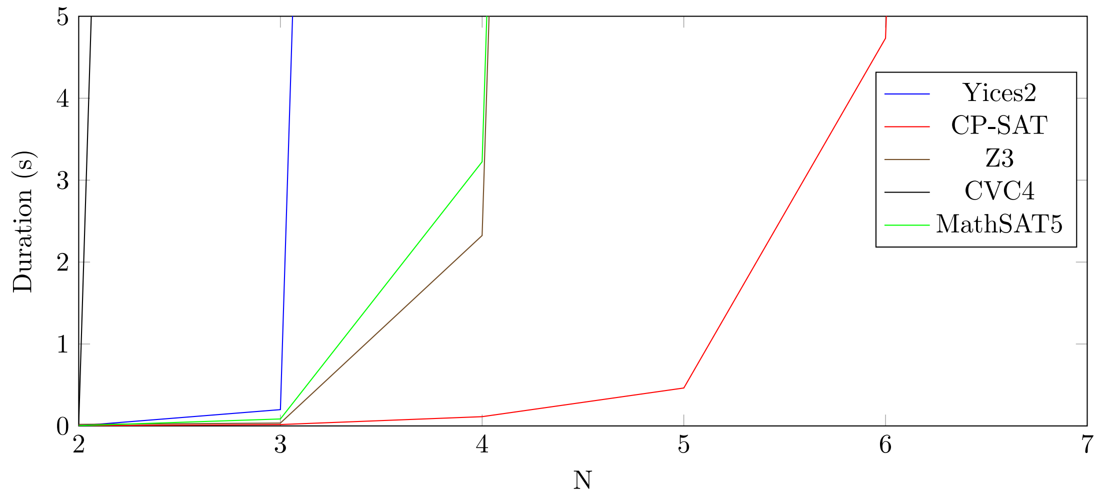

Figure 5.2: Average solve duration for different Sudoku sizes

Besides benchmarking for different solver sizes, we also measured the performance for finding all solutions of the Sudoku. To do this, we generate a set of Sudoku puzzles with various amount of solutions for $N = 3$. Then we solve these Sudoku puzzles and measure the solve duration. To retrieve all solutions we use the naive method as described in Chapter 4. The results are shown in Figure 5.3. For retrieving all solutions, the SMT solvers are significantly faster than CP-SAT.

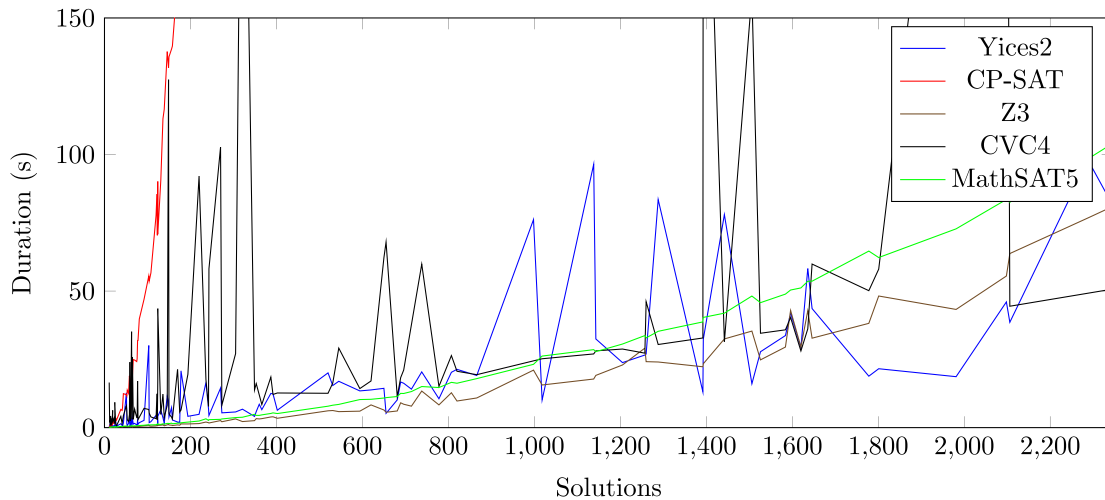

Figure 5.3: Finding all solutions of a 9x9 Sudoku problem (naive method)

This is expected since CP-SAT does not support incremental solving. Because CP-SAT lacks incremental solving, it has to solve from scratch for every solution while the SMT solvers can use the information found during previous solves to find the next solution.

5.2 Check SAT All Comparison

When using `AbsSolver#check_sat_all`, the interface tries to delegate finding all solutions to the solver as much as possible. MReason only supports this functionality for the CP-SAT and Yices2 solver at the moment. As shown in Figure 5.4, for SMT solvers there is not a big performance

increase compared to the naive method. This is because the only difference between the two implementations is that in `check_sat_all`, adding the blocking clause is delegated to the solver and the solver uses the same algorithm as the interface. For CP-SAT, however, there is a large difference between both methods. The solver is orders of magnitude faster in finding all solutions through `check_sat_all` compared to the naive method.

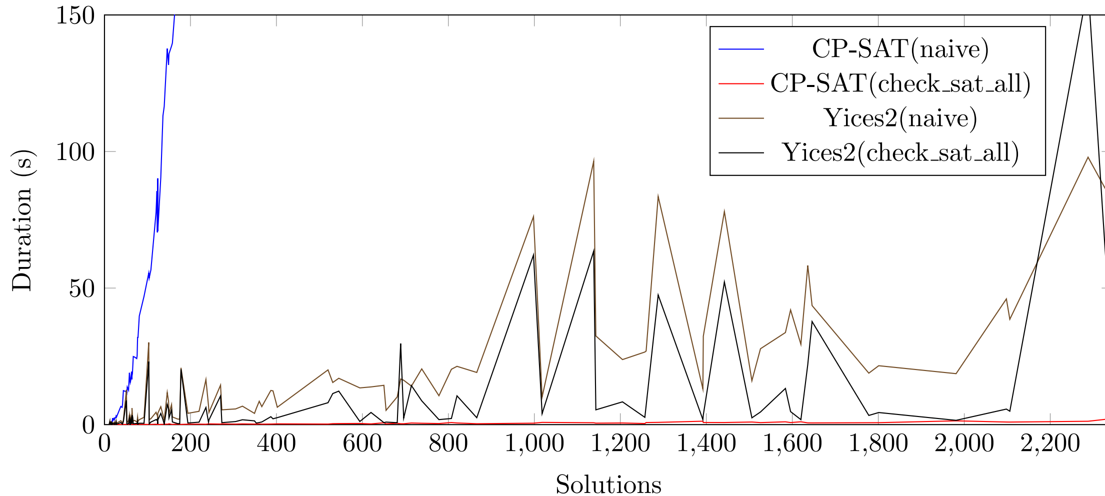

Figure 5.4: Finding all solutions of a 9x9 Sudoku problem

5.3 Solver API Comparison

Next, we measure the overhead of the interface itself. We do this by writing the model in both the MReason and CP-SAT API. Then we run the solver and measure the average solve time over five different Sudoku problems. We run two benchmarks, in the first benchmark we try to measure the overhead of storing the model at the interface level.

In the second benchmark we try to find out whether providing the domain to a variable when creating it is significantly faster than writing the domain as a constraint. Most MIP and CP solvers allow for specifying a domain when creating numeric variables. The MReason interface does not allow specifying this domain but the domain can be emulated by adding a lower and upper bound constraint on the variable. In this second measurement we want to investigate whether using this bounded constraint method is significantly better or worse than providing the domain directly when creating the variable. The results of these measurements are shown in Figure 5.5. Our measurements do not show a significant difference between using MReason and the CP-SAT interface directly. Similarly, the data shows no significant difference between using the bounded constraint method and the domain method. This data is not indicative of the overhead of MReason over all problems and all solvers, but it does provide evidence that the overhead of MReason is minimal. The source code for both CP-SAT benchmarks are shown in Appendix A.

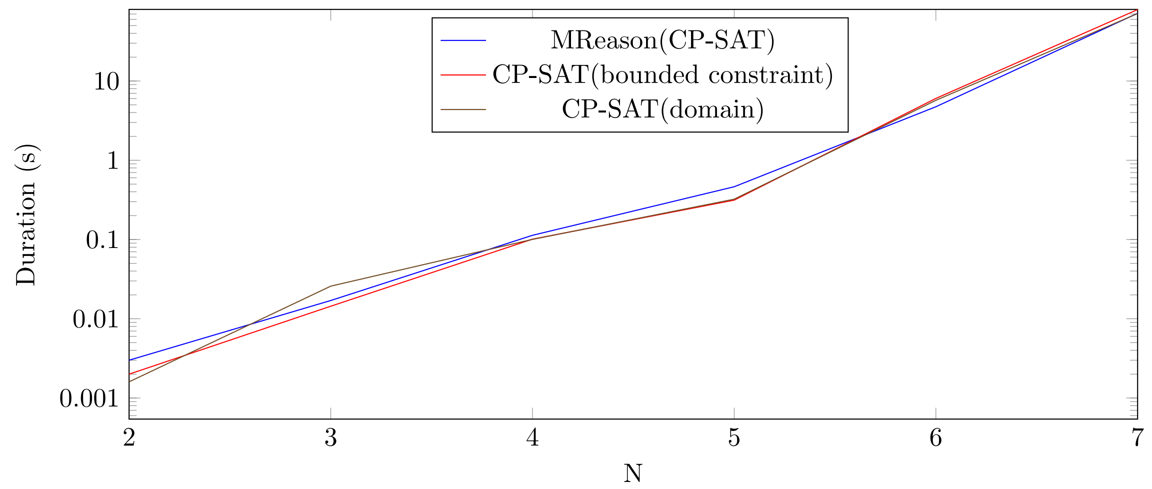

Figure 5.5: Average solve duration for different Sudoku sizes

Chapter 6

Future Work

In this thesis we mainly focused on adding support for as many different solvers as possible. We prioritized adding solver support to make the interface stable and show that it is solver agnostic. There are still a lot of aspects of MReason that can be improved in terms of usability, optimization and solver support. This chapter explains these aspects and discusses how they might be implemented.

6.1 Search Hints

Most CP solvers allow tuning the search process. Gecode supports the concept of post branching, which allows the user to modify the search tree after creating the model. Similarly, CP-SAT has the concept of solution hints, which can serve as a starting point for the solver to start searching. Both these solutions could be implemented in the interface through solution hints. Not all solvers, however, support these type of optimizations. Because they are optimizations and do not affect the outcome of the solving process this does not matter. The solver back-ends that do not support solution hints can just not support it and it will not affect their solving.

6.2 Operator Overloading

MReason allows creating terms from the `AbsSolver#make_symbol` and `AbsSolver#make_term` methods. This works well for creating simple models, but when building complex terms, calling these methods can create a lot of boilerplate code. Instead of using these term creation methods, it is also possible to build terms using operator overloaded terms. Operator overloading is a feature which is supported by most modern general purpose programming languages such as C++, Python, Rust, Kotlin and more. Take for example the following optimization problem:

$$\begin{aligned}
\max \quad & y && (6.1)\\
\text{subject to} \quad & y - x < 1 && (6.2)\\
& 3x + 2y < 12 && (6.3)\\
& 2x + 3y < 12 && (6.4)\\
& x, y \in \mathbb{Z} && (6.5)
\end{aligned}$$

This problem can be written in MReason as shown in Listing 6.1. Building the linear expressions require a lot of code, which makes the model difficult to read.

```
Solver s = ScipFactory::create();
Sort intsort = s->make_sort(SortKind::INT);
Term one = s->make_term(1, intsort);
Term two = s->make_term(2, intsort);
Term three = s->make_term(3, intsort);
Term twelf = s->make_term(12, intsort);
Term x = s->make_symbol("x", intsort);
Term y = s->make_symbol("y", intsort);

Term twoX = s->make_term(Op(Mult), two, x);
Term threeX = s->make_term(Op(Mult), three, x);
Term twoY = s->make_term(Op(Mult), two, y);
Term threeY = s->make_term(Op(Mult), three, y);

Term yMinX = s->make_term(Op(Minus), y, x);
Term threeXPlusTwoY = s->make_term(Op(Plus), threeX, twoY);
Term twoXPlusThreeY = s->make_term(Op(Plus), twoX, threeY);

Term con1 = s->make_term(Op(Le), yMinX, one);
Term con2 = s->make_term(Op(Le), threeXPlusTwoY, twelf);
Term con3 = s->make_term(Op(Le), twoXPlusThreeY, twelf);

s->assert_formula(con1);
s->assert_formula(con2);
s->assert_formula(con3);

Term obj = s->make_term(Op(Max), y);
s->add_objective(obj);
```

Listing 6.1: Non-operator overloaded

Using operator overloading it is possible to merge multiple `make_term` calls on operators into a single expression. The resulting model description is shown in Listing 6.2. With the operator overloading, it is a lot easier to read and understand the model.

```
Solver s = ScipFactory::create();
Sort intsort = s->make_sort(SortKind::INT);
Term one = s->make_term(1, intsort);
Term two = s->make_term(2, intsort);
Term three = s->make_term(3, intsort);
Term twelf = s->make_term(12, intsort);
Term x = s->make_symbol("x", intsort);
Term y = s->make_symbol("y", intsort);

Term con1 = s->make_term(y - x < one);
Term con2 = s->make_term(three * x + two * y < twelf);
Term con3 = s->make_term(two * x + three * y < twelf);

s->assert_formula(con1);
s->assert_formula(con2);
s->assert_formula(con3);

Term obj = s->make_term(Op(Max), y);
s->add_objective(obj);
```

Listing 6.2: Operator overloaded

Adding operator overloading support requires some design changes to the current MReason interface. Creating a term requires the underlying solver to be exposed to the interface back-end. Currently, this is done through the `AbsSolver` which is the receiver type for every `make_term` call. When using operator overloading, the left most operand is the receiver type, which is of type `AbsTerm`. `AbsTerm` does not hold a reference to the underlying solver object. To fix this, every `AbsTerm` should hold a reference to the `AbsSolver` it belongs to.

Adding this back-reference would also simplify retrieving the result after a solve. Currently, `AbsSolver#get_value` needs to be called before being able to retrieve the value. Getting an integer value for example can be done as follows: `s->get_value(intVar)->to_int()`. If the term holds a back-reference to the solver object, this could be simplified to: `intVar->to_int()`.

6.3 Default Constant Sorts

In the example shown in Listin 6.2 there is still a lot of boilerplate for creating constants and symbols. For symbols this is unavoidable; when creating a symbol, the user has to specify the Sort of the symbol. For constants it is possible to map a host language type to a `SortKind` in the interface as a default Sort. The C++ integer type; for example would map to `SortKind::Int`. Creating an integer constant could be simplified to `s->make_term(int_value)` when using this method. When combined with operator overloading, we can simplify the example in Listing 6.2 to Listing 6.3.

```
Solver s = ScipFactory::create();
Sort intsort = s->make_sort(SortKind::INT);
Term x = s->make_symbol("x", intsort);
Term y = s->make_symbol("y", intsort);

Term con1 = s->make_term(y - x < 1);
Term con2 = s->make_term(3 * x + 2 * y < 12);
Term con3 = s->make_term(2 * x + 3 * y < 12);

s->assert_formula(con1);
s->assert_formula(con2);
s->assert_formula(con3);

Term obj = s->make_term(Op(Max), y);
s->add_objective(obj);
```

Listing 6.3: Operator overloaded

6.4 Emulation

MReason currently supports solver features when they are supported by the solver itself. Sometimes it is possible to emulate solver features that are not supported by the solver. MReason already supports emulation in some very specific cases. For MIP models, for example, it is possible to use the $<$, $\geq$ and $>$ operators while this is not supported in the underlying solver API. An emulation technique that is sometimes supported in SAT/SMT interfaces is bit blasting. Bit blasting allows lowering bit vector operations in to Boolean level operations. This can be used to solve integer and bit vector problems on SAT solvers.

6.5 Global Constraints

The current interface does not support any global constraints that usually exist in CP solvers, except for the `alldifferent` constraint, which is equivalent to the `distinct` operator in SMT solvers. Even though global constraints are generally only supported by CP solvers, they can be emulated in the existing interface for solvers which do not support them. Further research is required to evaluate which global constraints should be added and how to properly emulate them.

6.6 All Solutions Limit

MReason only supports finding a single solution using `AbsSolver#check_sat` or finding all solutions using `AbsSolver#check_sat_all`. It is not possible to limit the amount of solutions even though it is supported by most solver APIs. Support for this can be added by overloading the `check_sat_all` method in the front-end with an additional parameter that expresses the limit to the amount of solutions to find.

6.7 Solver Support

Currently, MReason support nine MIP solvers, five SMT solvers and one CP solver. At the moment, there is no support for SAT solvers, but adding support for SAT solvers should be trivial since they are very similar to SMT solvers. To make SAT solvers a useful addition to MReason, emulation is required. Modeling problems in CNF is very difficult for humans. A library like PBLib [11] can be used to encode Pseudo-Boolean formulas into a CNF, which would allow users to model for SAT solvers using Pseudo-Boolean constraints.

Besides all the solvers mentioned in Chapter 4, MReason has implementations for two other solvers; BitWuzla [35], an SMT solver and Gecode [36], a CP solver. Implementations for these solvers are not complete or have limited functionality.

6.8 Performance Evaluation

In Chapter 5 we already evaluated many performance aspects of MReason. We, however, did not evaluated the performance of the Python interface in comparison with the C++ interface and other Python interfaces. These benchmarks are useful when deciding whether to use the C++ or Python interface when high performance is required.

Besides evaluating the raw performance of MReason it would also be interesting to measure the memory overhead when using MReason. Because the CP back-end, MIP back-end and `LoggingSolver` all store the model at the interface level, the memory usage should be higher. This could be a problem for large models or when running MReason on an embedded system, where the amount of memory might be limited.

6.9 Unified LoggingSolver

Currently, the `LoggingSolver` only works for the SMT back-end implementations. This is because, compared to the CP and MIP implementations, the SMT back-end implementations do not store the AST at the interface level. To be able to migrate between MIP, SMT, and CP solvers at runtime, support should be added for the CP and MIP solvers. More research needs to be done in figuring out how to unify the `LoggingSolver` across all types of solvers back-ends.

6.10 Numeric Variable Domains

In Chapter 5, we mentioned that we measured the performance between providing the numeric variable domain when creating the variable, compared to writing the domain as a constraint. In this measurement, we found that it makes no difference for the CP-SAT solver. This does not mean that other solvers will not be affected by this method. Therefore, the interface should allow the user to specify the domain when creating a numeric variable. In SMT solvers, which typically do no support this in their interfaces the constraint method can still be used to specify the domain.

Chapter 7

Conclusion

In this master thesis we have shown an implementation of a solver agnostic machine reasoning interface that supports incremental solving to answer the research question on how to build a C++ interface that interacts with SAT, MIP, SMT and CP solvers (RQ1). We did this by introducing a C++ interface, called MReason, which supports MIP, SMT and CP solvers. MReason currently does not support SAT solvers, but adding support for them should not pose a big challenge because they are very similar to SMT solvers. Lastly, we wanted to find out whether there is any significant overhead of using such an unifying solver interface compared to directly using the solver APIs (RQ2). In this research we investigated this by comparing the MReason API performance to the CP-SAT API performance. From this comparison, we have found no significant difference in the solving performance of both APIs. This indifference in performance makes MReason a viable option for solving CSPs over regular solving APIs, even when high performance is required. Further research, however, is required to improve the usability of MReason and add support for more solver aspects, like global constraints.

Bibliography

[1] T. Jim and J. Palsberg, "Type inference in systems of recursive types with subtyping," 1999. 1

[2] G. T. Peter J. Stuckey, Kim Marriott, "The minizinc handbook 2.1.1 section," 2020. 1

[3] K. L. McMillan, "Interpolation and sat-based model checking," in *Computer Aided Verification* (W. A. Hunt and F. Somenzi, eds.), (Berlin, Heidelberg), pp. 1–13, Springer Berlin Heidelberg, 2003. 1

[4] F. Haedicke, B. Alizadeh, G. Fey, M. Fujita, and R. Drechsler, "Polynomial datapath optimization using constraint solving and formal modelling," in *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 756–761, 2010. 1

[5] G. T. Peter J. Stuckey, Kim Marriott, "The minizinc handbook 2.5.0 section," 2020. 1

[6] D. Berthier, "Pattern-based constraint satisfaction and logic puzzles," 11 2012. 1

[7] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "Minizinc: Towards a standard cp modelling language," in *Principles and Practice of Constraint Programming – CP 2007* (C. Bessière, ed.), (Berlin, Heidelberg), pp. 529–543, Springer Berlin Heidelberg, 2007. 1

[8] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2010. 1

[9] M. Mann, A. Wilson, C. Tinelli, and C. Barrett, "Smt-switch: a solver-agnostic c++ api for smt solving," 07 2020. 1

[10] S. Ranise and C. Tinelli, "The SMT-LIB Standard: Version 1.2," tech. rep., Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`. 1

[11] T. Philipp and P. Steinke, "Pblib–a library for encoding pseudo-boolean constraints into cnf," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 9–16, Springer, 2015. 4, 24

[12] S. Böhme, *Proving theorems of higher-order logic with SMT solvers.* PhD thesis, Technische Universität München, 2012. 5

[13] M. Davis, "Hilbert's tenth problem is unsolvable," *The American Mathematical Monthly*, vol. 80, no. 3, pp. 233–269, 1973. 5

[14] J. McCarthy, "Towards a mathematical science of computation," in *IFIP Congress*, pp. 21–28, 1962. 6

[15] J.-C. Régin, "A filtering algorithm for constraints of difference in csps," 01 1994. 6

[16] S. D. Prestwich, "Cnf encodings.," *Handbook of satisfiability*, vol. 185, pp. 75–97, 2009. 8

[17] R. Fourer, D. M. Gay, and B. W. Kernighan, "A modeling language for mathematical programming," *Management Science*, vol. 36, no. 5, pp. 519–554, 1990. 8

[18] L. Perron and V. Furnon, "Or-tools." 8, 14, 15

[19] H. Riener, F. Haedicke, S. Frehse, M. Soeken, D. Große, R. Drechsler, and G. Fey, "Metasmt: Focus on your application and not on solver integration," *Int. J. Softw. Tools Technol. Transf.*, vol. 19, p. 605–621, Oct. 2017. 8

[20] F. Boussemart, C. Lecoutre, G. Audemard, and C. Piette, "Xcsp3-core: A format for representing constraint satisfaction/optimization problems," *arXiv preprint arXiv:2009.00514*, 2020. 8

[21] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, *et al.*, "Extensible markup language (xml) 1.0," 2000. 8

[22] Y. Yu, P. Subramanyan, N. Tsiskaridze, and S. Malik, "All-sat using minimal blocking clauses," in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pp. 86–91, IEEE, 2014. 12

[23] johnjforrest, S. Vigerske, T. Ralphs, L. Hafer, jpfasano, H. G. Santos, M. Saltzman, h-i gassmann, B. Kristjansson, and A. King, "coin-or/clp: Version 1.17.6," Apr. 2020. 14

[24] johnjforrest, S. Vigerske, H. G. Santos, T. Ralphs, L. Hafer, B. Kristjansson, jpfasano, EdwinStraver, M. Lubin, rlougee, jpgoncal1, h-i gassmann, and M. Saltzman, "coin-or/cbc: Version 2.10.5," Mar. 2020. 14

[25] G. Gamrath, D. Anderson, K. Bestuzheva, W.-K. Chen, L. Eifler, M. Gasse, P. Gemander, A. Gleixner, L. Gottwald, K. Halbig, G. Hendel, C. Hojny, T. Koch, P. Le Bodic, S. J. Maher, F. Matter, M. Miltenberger, E. Mühmer, B. Müller, M. E. Pfetsch, F. Schlösser, F. Serrano, Y. Shinano, C. Tawfik, S. Vigerske, F. Wegscheider, D. Weninger, and J. Witzig, "The SCIP Optimization Suite 7.0," March 2020. 14

[26] L. Gurobi Optimization, "Gurobi optimizer," 2020. 14

[27] IMB, "Ibm cplex optimizer," 2020. 14

[28] R. Ashford, "Mixed integer programming: A historical perspective with xpress-mp," *Annals of Operations Research*, vol. 149, no. 1, p. 5, 2007. 14

[29] A. Makhorin, "Glpk (gnu linear programming kit)."
Available at `http://www.gnu.org/software/glpk/glpk.html`. 14

[30] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2 , btormc and boolector 3.0," in *Computer Aided Verification* (H. Chockler and G. Weissenbacher, eds.), (Cham), pp. 587–595, Springer International Publishing, 2018. 15

[31] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Proceedings*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pp. 171–177, 2011. 15

[32] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (N. Piterman and S. A. Smolka, eds.), (Berlin, Heidelberg), pp. 93–107, Springer Berlin Heidelberg, 2013. 15

[33] B. Dutertre, "Yices 2.2," in *Computer Aided Verification* (R. Biere, Arminand Bloem, ed.), (Cham), pp. 737–744, Springer International Publishing, 2014. 15

[34] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008. 15

[35] A. Niemetz and M. Preiner, "Bitwuzla at the smt-comp 2020," *arXiv preprint arXiv:2006.01621*, 2020. 24

[36] Gecode Team, "Gecode: Generic constraint development environment," 2021. Available from `http://www.gecode.org`. 24

Appendix A

Sudoku Model Source Code

```cpp
std::vector<std::vector<Term>> buildSudokuModel(
    Solver& s,
    const std::vector<std::vector<int>>& puzzle
) {
    Sort intSort = s->make_sort(INT);
    std::vector<std::vector<Term>> cells(puzzle.size(), std::vector<Term>(puzzle.size()));
    for (int i = 0; i < puzzle.size(); i++) {
        for (int j = 0; j < puzzle.size(); j++) {
            Term t = s->make_symbol(std::to_string(i) + "," + std::to_string(j), intSort);
            cells[i][j] = t;
            if (puzzle[i][j] != 0) {
                Term eq = s->make_term(Op(Equal), t, s->make_term(puzzle[i][j], intSort));
                s->assert_formula(eq);
            } else {
                Term min = s->make_term(Op(Gt), t, s->make_term(0, intSort));
                Term max = s->make_term(Op(Le), t, s->make_term((int)puzzle.size(), intSort));
                s->assert_formula(min);
                s->assert_formula(max);
            }
        }
    }
    for (int i = 0; i < puzzle.size(); i++) {
        Term rowDif = s->make_term(Op(Distinct), cells[i]);
        s->assert_formula(rowDif);
        std::vector<Term> column;
        column.reserve(puzzle.size());
        for(int j = 0; j < puzzle.size(); j++) {
            column.emplace_back(cells[j][i]);
        }
        Term columnDif = s->make_term(Op(Distinct), column);
        s->assert_formula(columnDif);
    }
    for(int i = 0; i < puzzle.size(); i += (int) sqrt(puzzle.size())) {
        for(int j = 0; j < puzzle.size(); j += (int) sqrt(puzzle.size())) {
            std::vector<Term> sub;
            sub.reserve(puzzle.size());
            for(int k = i; k < i + sqrt(puzzle.size()); k++) {
                for(int l = j; l < j + sqrt(puzzle.size()); l++) {
                    sub.emplace_back(cells[k][l]);
                }
            }
            Term subDif = s->make_term(Op(Distinct), sub);
            s->assert_formula(subDif);
        }
    }
    return cells;
}
```

Listing A.1: MReason Sudoku

```
std::vector<std::vector<IntVar>> buildSudokuModel(
    CpModelBuilder& cp_model,
    const std::vector<std::vector<int>>& puzzle
) {
    const operations_research::Domain domain(INT_MIN, INT_MAX);
    std::vector<std::vector<IntVar>> cells(puzzle.size(), std::vector<IntVar>(puzzle.size()));
    for (int i = 0; i < puzzle.size(); i++) {
        for (int j = 0; j < puzzle.size(); j++) {
            const IntVar t = cp_model.NewIntVar(domain).WithName(std::to_string(i) + "," + std
    ::to_string(j));
            cells[i][j] = t;
            if (puzzle[i][j] != 0) {
                cp_model.AddEquality(t, puzzle[i][j]);
                break;
            } else {
                cp_model.AddGreaterThan(t, 0);
                cp_model.AddLessOrEqual(t, puzzle.size());
            }
        }
    }
    for (int i = 0; i < puzzle.size(); i++) {
        cp_model.AddAllDifferent(cells[i]);
        std::vector<IntVar> column;
        column.reserve(puzzle.size());
        for(int j = 0; j < puzzle.size(); j++) {
            column.emplace_back(cells[j][i]);
        }
        cp_model.AddAllDifferent(column);
    }
    for(int i = 0; i < puzzle.size(); i += (int) sqrt(puzzle.size())) {
        for(int j = 0; j < puzzle.size(); j += (int) sqrt(puzzle.size())) {
            std::vector<IntVar> sub;
            sub.reserve(puzzle.size());
            for(int k = i; k < i + sqrt(puzzle.size()); k++) {
                for(int l = j; l < j + sqrt(puzzle.size()); l++) {
                    sub.emplace_back(cells[k][l]);
                }
            }
            cp_model.AddAllDifferent(sub);
        }
    }
    return cells;
}
```

Listing A.2: CP-SAT Sudoku(bounded constraint)

```
std::vector<std::vector<IntVar>> buildSudokuModel(
    CpModelBuilder& cp_model,
    const std::vector<std::vector<int>>& puzzle
) {
    const operations_research::Domain domain(0, puzzle.size());
    std::vector<std::vector<IntVar>> cells(puzzle.size(), std::vector<IntVar>(puzzle.size()));
    for (int i = 0; i < puzzle.size(); i++) {
        for (int j = 0; j < puzzle.size(); j++) {
            const IntVar t = cp_model.NewIntVar(domain).WithName(std::to_string(i) + "," + std
    ::to_string(j));
            cells[i][j] = t;
            if (puzzle[i][j] != 0) {
                cp_model.AddEquality(t, puzzle[i][j]);
            }
        }
    }
    for (int i = 0; i < puzzle.size(); i++) {
        cp_model.AddAllDifferent(cells[i]);
        std::vector<IntVar> column;
        column.reserve(puzzle.size());
        for(int j = 0; j < puzzle.size(); j++) {
            column.emplace_back(cells[j][i]);
        }
        cp_model.AddAllDifferent(column);
    }
    for(int i = 0; i < puzzle.size(); i += (int) sqrt(puzzle.size())) {
        for(int j = 0; j < puzzle.size(); j += (int) sqrt(puzzle.size())) {
            std::vector<IntVar> sub;
            sub.reserve(puzzle.size());
            for(int k = i; k < i + sqrt(puzzle.size()); k++) {
                for(int l = j; l < j + sqrt(puzzle.size()); l++) {
                    sub.emplace_back(cells[k][l]);
                }
            }
            cp_model.AddAllDifferent(sub);
        }
    }
    return cells;
}
```

Listing A.3: CP-SAT Sudoku(domain)