Encoding The Lexicographic Ordering Constraint in Satisfiability Modulo Theories

Hani Abdalla Muftah Elgabou

MSc by Research

University of York
Computer Science

February 2015

Dedication

To my Mother, my Wife and my Daughters

Abstract

In this thesis we present nine different SMT encodings for the Lexicographic Ordering Constraint (*Lex*), These constraints are helpful in breaking some kinds of symmetries in decision problems. The encodings are drawn from the constraint solving literature or is a variant of such. This thesis aims to serve as a single source for all known to date encodings for the lexicographic ordering constraint.

For the purpose of benchmarking, the encodings are translated into an SMT suitable form. We have done this using two methods, the first is by directly translating the encodings into SMT using C# code. The second starts by rewriting the encodings in MiniZinc language, flattening them into FlatZinc instances, then using a tool called fzn2smt to translate them to SMT. We evaluated the encodings on a suite of instances of the Social Golfer problem and the Balanced Incomplete Block Design problem, both are well known for their highly symmetric models. We tried to run inference capability tests using unsatisfiable instances of *Lex* between two long vectors, also by trying to find all solutions for instances of the BIBD problem. We used the SMT solvers Yices2 and Z3 to benchmark the encodings.

Our results show that what we called the Recursive OR encoding performed better than all the other encoding on most of the instances, but it was notably worse on other instances. This behaviour is roughly shared by some of the other encodings, it shows that different encodings perform differently on different problems. The results also show that, in many cases, not having any symmetry braking using either of the nine encodings performed surprisingly better.

Contents

List of Figures

List of Tables

Acknowledgements

All praise to Allah, the lord of the worlds,
the most beneficent and the most merciful.

I wish to express my sincere gratitude to my supervisor Dr. Alan M. Frisch for his patience and all the guidance and encouragement he provided me throughout this research. I would like to thank my assessor Dr. James Cussens for his support and feedback. I wish also to thank all staff and students at the University of York for the great friendly environment and for providing the facilities. With much appreciation, I recognize that this work would not have been possible without the financial assistance of the Libyan people, represented by the Libyan embassy in London.

Finally and most importantly, I would like to express my love and gratitude to Najwa, my wife, for her continued encouragement and patience.

Declaration

I declare that the research described in this thesis is original work, which I undertook at the University of York during 2013 - 2015. Except where stated, all of the work contained within this thesis represents the original contribution of the author.

Chapter 1

Introduction

Modern SAT solvers are essential tools in many applications today, including solving problems related to scheduling, verification, circuit design and more. Before to be solved, a problem need to be modelled as a Boolean Satisfiability Problem, in which the problem is translated to a group of Boolean constraints. These constraints are represented by groups of Boolean formulas called Conjunctive Normal Form or CNF. The increasing complexity of some of today's combinatorial problems makes them difficult to be represented by only using SAT's Boolean formulas . Satisfiability Modulo Theories (SMT) has been introduced to overcome this limitation [4] [21], where problems can be encoded using logical formulas of combinations of atomic propositions and atomic expressions in one or more theories T. The theory part in SMT formulas enables them to naturally describe problems related to any of the SMT supported theories, like arithmetic, arrays and bit-vectors. SMTLIB is the standard modelling language in SMT.

The phenomenon of symmetry arises in many constraint satisfaction problem models. A common form of symmetry is the interchangeability between elements of sets of variables and the corresponding sets of values. An example is the ability to swap any two rows or columns in a Latin Square while still preserving validity of its rules. Breaking symmetry reduces the search space, which could improve performance [49] [15].

The *Lex* ordering constraint enforces lexicographic ordering between two vectors of variables, which makes it useful in breaking symmetry between rows and between columns of a matrix of decision variables [53] [23] [3]. For example, enforcing *Lex* between each row and its next in a Latin Square eliminates the interchangeability between the rows, the same applies for the columns.

1.1 Motivation and Aims

A recent study demonstrated that SMT solvers have a competitive performance in solving Constraint Satisfaction Problems (CSP). In that study, Bofill *et al* [12] built a tool called fzn2smt to translate CSP instances expressed in Flatzinc into the SMT language SMTLIB1.2 standard. Then they solved these instances using the Yices1 SMT solver [22]. Yices1, in general, performed better than some well known constraint solvers. This performance makes encoding CSP problems in SMT a promising research line. Bofill's results show that fzn2smt and Yices1 underperformed on instances that involve global constraints, some of them have the lexicographic ordering constraint. The aim of this thesis is to find whether there are better ways to encode the lexicographic constraint in SMT. We will try this by studying and benchmarking different SMT encodings of the lexicographic ordering constraint and compare their performance. We will also study the effect that Bofill's translator has on these encodings when it is used to produce them.

1.2 Approach and Results

This thesis presents nine different encodings of the lexicographic ordering constraint.

1. The AND encoding [27]
2. The AND CSE encoding
3. The OR encoding [27]
4. The OR CSE encoding
5. The Recursive OR encoding [30]
6. The Arithmetic encoding [27]
7. Harvey's encoding [27]
8. The Alpha encoding [30]
9. The AlphaM encoding [1]

Seven of the encodings are obtained from the literature then for the purpose of benchmarking translated into an SMT suitable form. We have done this using two methods, the first is by directly translating the encodings into SMT. The second follows Bofill's method of translating Minizinc instances to SMT using MiniZinc's mzn2fzn and the fzn2smt tool. It starts by rewriting the encodings in MiniZinc language, flattening them into FlatZinc instances using MiniZinc's mzn2fzn tool, then translating them to SMT using fzn2smt. We evaluate the encodings on a suite of instances of the Social Golfer problem and Balanced Incomplete Block Design. Both are well known for their highly symmetric models. We also tried to run inference capability tests using unsatisfiable instances of *Lex* between two long vectors and by trying to find all solutions for some instances of the Balanced Incomplete Block Design problem. We used Yices2 [22] and Z3 [17] SMT solvers to benchmark the encodings.

The results in Chapter 4 are for the benchmarking suite problems for each of the SMT solvers. We evaluated the performance of each encoding by recording the time it takes the solver to solve its instances. To find the best possible typical timing for each instance, we ran 30 different samples per instance. These samples were generated by randomly arranging the lines of code of the instances.

To test the effect of each encoding on the benchmarking problems, we also ran tests on all the instances without including any symmetry braking constraints (No Lex). Although many results show that the No Lex has better timings than all of the encodings, the picture would be completely different in cases of unsatisfible instances or when searching for all possible solutions, as we will see in the all solutions benchmarks section. In the cases of unsatisfible instances and all solutions, a solver needs to explore all possible paths of the search tree to find all solutions or to prove none is exists. Symmetry breaking aims to reduce number of these paths and makes the solving process runs faster. An example to this effect is clear in the results of the unsatisfible instance of 8-4-4 in the result tables for the BIBD problem.

1.3 Contributions and Results

This thesis is the first collection of all known to date encodings for the lexicographic ordering constraints. We translated and evaluated these encodings in SMT. Our benchmarks results show that, typically, formula size could greatly affect the solving time. Our results also show that different encoding perform differently on different instances, this implies that using specialized algorithms to propagate *Lex* might be better than decomposing it into many smaller constraints. The results also proved that Bofill *et al* would have generally got better performances if they had used a direct translation to SMT where it is possible instead of using the fzn2smt. One surprising finding in this research, is that in the long vectors test, which are large instances of pure *Lex*, the DNF OR encoding performed better than the CNF AND encoding. CNF is the modelling language for SAT solvers, which are at the heart of the SMT solving process.

Chapter 2

Background

2.1 Boolean Satisfiability

The Boolean Satisfiability Problem (SAT) is the problem of finding one or more assignments for a propositional formula which evaluate it to true, or proving no such assignment exists.

The kind of propositional formulas that Boolean satisfiability deals with are formulas composed of Boolean variables connected with relations of Boolean algebra. The basic forms of these relations are AND, OR and NOT ($\wedge, \vee, \neg$).

Let A, B and C be Boolean variables. The following three formulas are examples of propositional formulas.

$$A \vee B \tag{2.1}$$

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

$$\neg C \tag{2.3}$$

A single Boolean variable is a formula and a formula with no variables is called an empty formula.

Let α, β be propositional formulas, from the above examples we can note that all the following are also propositional formulas:

$$\alpha \wedge \beta$$
$$\alpha \vee \beta$$
$$\neg \alpha$$

Throughout this thesis we will use *formula* to refer to a propositional formula and *variable* for Boolean variable .

Finding an assignment for a Boolean formula or proving none exists is the main goal of the Boolean SAT. A valid assignment for a formula is the set of $true$ and $false$ values that can be assigned to all of its variables which evaluates the formula to $true$. Formulas that have one or more valid assignment are denoted satisfiable (Sat), while the ones with no such assignment are called unsatisfiable (Unsat). Here are some examples of formulas and their satisfying assignments:

$$(A \vee \neg B) \wedge (\neg A \vee \neg B) \quad A = true\ and\ B = false$$
$$(A \vee \neg B) \wedge (\neg A \vee \neg B) \quad A = false\ and\ B = false$$
$$(\neg A \vee B) \wedge (A \vee B) \wedge \neg B \quad Unsat\ (has\ no\ satisfying\ assignment)$$

2.1.1 Conjunctive Normal Form

Conjunctive Normal Form or CNF can be regarded as an input language to almost all current SAT solvers. Before solving a combinatorial problem using SAT, the problem needs to be translated or encoded in Conjunctive Normal Form.

Conjunctive Normal Form is a conjunction of disjunctions. It is conjunctions of clauses in which each clause is formed of disjunction of literals. A literal is a variable in one logical state. For any variable A, A is a literal and $\neg A$ is regarded as a different literal.

The following is an example of a formula in CNF:

$$(A \vee B) \wedge (A \vee \neg B \vee \neg C) \wedge D$$

In the above formula, A, B, C and D are literals, so are their negations. $(A \vee B)$ is a clause and also $(A \vee \neg B \vee \neg C)$. D at the end of the formula is called a unit clause, which is a clause that contains only one literal. We will see later how unit clauses are very helpful in solving CNF formulas.

Using CNF has it is advantages, among them are, transforming any propositional formula to CNF is relatively easy, can be done in a linear time and produces a formula of a linear size compared to the original but with more variables [16]. Also, the structure of the encoding facilitated relatively small but efficient algorithm to solve SAT problems, namely algorithms based on the DPLL algorithm [16].

2.2 SAT Solvers

SAT problem was the first problem to be proven to be NP-Complete [14]. A consequence of SAT's NP-Completeness is that there are no known algorithms that can solve worst-case instances of SAT in a feasible time. However, the importance of SAT solving to a wide range of applications made it an active research field during the last decade. The advances in modern SAT solvers made them efficient in solving many difficult real world problems on different domains, this success drove the development of SAT solvers which in turn inspired more applications [44].

2.2.1 DPLL algorithm

most modern SAT solvers are based on the DPLL (Davis, Putnam, Logemann and Loveland) algorithm, which is a Search-Backtracking algorithm presented by Davis *et al* in 1962 [16].

DPLL algorithm works only on formulas in CNF. To solve a CNF formula, DPLL first tries to simplify the input CNF, then selects one of its literals and assign it a value, either $true$ or $false$, and checks whether there is a conflict. A conflicting value is a value that evaluates a formula to $false$. When a conflict arose, DPLL backtracks and flips the value of the literal. In case of no conflicts DPLL picks another literal, assigns it a value and repeats the same operation. DPLL works recursively though all the literals in the formula, at the same time, trying to resolve the CNF on each recursion. DPLL terminates with two possible outcomes, the first, it manages to assign values to all of the atoms in the CNF, which proves the satisfiability of the formula, or it hits a conflict which cannot be resolved by backtracking, in this case the formula is proven to be unsat.

To simplify a formula, DPLL uses two formula resolution techniques, Unit Propagation or Boolean constraints Propagation (BCP) and the Pure Literal Rule. In both the algorithm tries to reduce the input formula as much as possible before starting the search. Unit propagation relies on two implications of a unit clause in a formula, for example, let literal C be a unit clause in a formula, removing all other clauses that contain C will not change the satisfiability state of the formula, also removing any occurrence of the complement of C from the formula has no effect on its satisfiability. To demonstrate this here is an example:

$$(A \vee B \vee C) \wedge (\neg A \vee B) \wedge (\neg B \vee \neg C) \wedge A \tag{2.4}$$

$$B \wedge (\neg B \vee \neg C) \wedge A \tag{2.5}$$

$$B \wedge \neg C \wedge A \tag{2.6}$$

In (2.4), A forms a unit clause, so $(A \vee B \vee C)$ and $\neg A$ were eliminated from the formula. This resulted in B being the new unit clause as shown in formula (2.5), so we propagate B. The technique is to keep applying unit propagation till no chances of further simplifying the formula. (2.6) can not be simplified again, because all clauses are unit, so from (2.6), the satisfying assignment for the formula is $B = true$, $C = False$ and $A = true$. If the propagation resulted in an empty clause then the formula is proven to be unsatisfiable as shown by the following.

$$(A \vee \neg B) \wedge (A \vee B) \wedge \neg A \tag{2.7}$$

$$Propagate\ \neg A,\ (\neg B) \wedge (B) \wedge \neg A \tag{2.8}$$

$$Propagate\ B,\ (Empty\ Clause) \wedge (B) \wedge \neg A \tag{2.9}$$

As for the Pure Literal Rule, a pure literal is a Boolean variable that only appears in a single Boolean state throughout its occurrences in a CNF formula. The Pure Literal Rule states that a CNF formula remains equisatisfiable when setting all of its pure literals to $True$. This property implies that removing all clauses containing pure literals also retains the equisatisfiability. For example, consider the following formula.

$$(A \vee B) \wedge (A \vee \neg B) \wedge \neg B \tag{2.10}$$

$$A\ is\ Pure,\ \ A = True \tag{2.11}$$

$$A \wedge \neg B \tag{2.12}$$

Both (2.10) and (2.12) are logically equivalent and their only satisfiable assignment is $A = True$ and $B = False$

In the next pseudo code for the DPLL algorithm, $PureLiteral(\phi)$ and $UnitPropagate(\phi)$ will apply the Pure Literal rule and Unit Propagation on the input formula ϕ. $PickLiteral(\phi)$ picks a different literal l from ϕ on each run. On a successful pick, l gets assigned a value, a failure means that there are no more literals to choose, i.e. all literals in ϕ have been successfully assigned a value and a satisfiable assignment has been found.

Algorithm 1 : DPLL (ϕ)

Require: CNF Formula: ϕ
Ensure: Satisfiabilty of ϕ: $(True, False)$

```
φ = PureLiteral(φ)
φ = UnitPropagate(φ)
if □ ∉ φ (□ = An empty clause) then
    if PickLiteral(φ) then
        if DPLL(φ ∧ l) (Assigns True to l in φ) then
            return True
        end if
        if DPLL(φ ∧ ¬l) (Assigns False to l in φ) then
            return True
        end if
    else
        return True
    end if
end if
return False
```

2.2.2 SAT Solvers Enhancements

The importance of SAT solving to many applications motivated an intensive research efforts to improve the DPLL algorithm [45] [46]. The performance of Modern SAT solvers has been greatly increased by fine tuning DPLL and introducing new techniques, such as Optimized BCP, Conflict Analysis, Clause Learning, Heuristic Strategies and Restarts, in addition to efforts to increase performance by using SAT solvers in parallel to solve problems [61] [34]. Here we will briefly discuss some of the important enhancements.

- Optimized BCP

When solving SAT problems most of the SAT Solvers' running time is spent on BCP, that is the case because whenever the DPLL algorithm assigns a value to a literal the propagation algorithm is continuously revisiting all the clauses checking whether any of them have become unit clauses. To ease this overhead Moskewicz *et al* introduced the Watched 2-Literals method which greatly improved the performance of SAT solving [46].

- Conflict Analysis and Clause Learning

  Most of state-of-the-art SAT solvers are Conflict Driven Clause Learning solvers (CDCL), where the search is guided by the analysis of conflicts to produce learned clauses and strategies to decide where to Back-Jump from a conflict (Non-Chronological-Backtracking) [62]. Conflict analysis is also used to calculate when it is a good time to decide that the current solving efforts are unfruitful then restart. Restarts throw away all current assignments but preserve some of the useful learned clauses. These methods are used to prune the search space and guide the solver to more promising paths to find a SAT assignment.

- Heuristic Strategies

  Heuristic Strategies concerned with the question; After propagation which literal to pick next to assign a value?. This choice could greatly affect the solving process, making bad ones could lead the solver to endless paths of unsat assignments. There are different heuristic strategies around but all of them depend on conflict analysis [62].

2.2.3 Applications of SAT Solvers

SAT solving has provided solutions for many applications, these include; software and hardware testing [39] [43], model checking and design verification and debugging [11] [55], in fact SAT solvers works as a backbone for some of today's digital circuits design applications [52] [42].

Take Model Checking application as an example, most of the SAT based model checking applications rely on the concept of a Safety Properties [44] [11] [37], which is a set of Boolean constraints that must be satisfied on all states of the model. To check a model SAT solvers are used to try to find a complement of its Safety Properties, which is basically trying to find any assignment that makes the model fail.

Although Verification and Model Checking are the main application areas of SAT solvers, they also show promising performances in other real world domains, such as scheduling, planning [60] and optimisation using Max-SAT [28].

2.3 Satisfiability Modulo Theories

As SAT solvers started to gain popularity in more applications, the need for more expressive modelling language beyond the SAT's CNF became more apparent. Some problems could be naturally represented by means of one or more background theories. For example, the arithmetic elements in a code, the physical properties of a system or the states of a model could be represented by the theory of linear arithmetic [10]. Surely some of these problems could still be modelled using CNF, but doing this could result in very large and tedious formulas and with a different abstraction level from the original model [51]. Consider the formula $A + B = 3$ and $A - B = 1$, where $1 \leq A \leq 3$ and $1 \leq B \leq 3$ for integers A and B. A possible CNF encoding for the formula $(A+B=3) \wedge (A-B=1)$ is:

$$
\begin{aligned}
& A1 = true\ for\ A = 1,\ A2 = true\ for\ A = 2\ and\ A3 = true\ for\ A = 3 \\
& B1 = true\ for\ B = 1,\ B2 = true\ for\ B = 2\ and\ B3 = true\ for\ B = 3 \\
& Possible\ values\ are\ \{1,2,3\}.\ Encoded\ by\ the\ constraints\ in\ (2.13) \\
& Conflicting\ values\ for\ (A,B)\ are\ (1,1),(2,2),(3,3),(3,1),(1,3),(2,3),(3,2) \\
& The\ conflicting\ values\ are\ encoded\ by\ the\ constraints\ in\ (2.14)\ and\ (2.15) \\
& The\ CNF:
\end{aligned}
$$

$$(A1 \vee A2 \vee A3) \wedge (B1 \vee B2 \vee B3)\ \wedge \tag{2.13}$$

$$(\neg A1 \vee \neg B1) \wedge (\neg A2 \vee \neg B2) \wedge (\neg A3 \vee \neg B3) \wedge (\neg A3 \vee \neg B1)\ \wedge \tag{2.14}$$

$$(\neg A1 \vee \neg B3) \wedge (\neg A2 \vee \neg B3) \wedge (\neg A3 \vee \neg B2) \tag{2.15}$$

There are other ways to encode the previous formula in CNF [29] [58] [56], but introducing more variables other than A and B is unavoidable in all of them. If we repeat the same process but with domains of $-10 \leq A \leq 10$ and $-10 \leq B \leq 10$, we would need 21 auxiliary variables in addition to a special algorithm to figure out all the conflicting pairs of values. The answer to this problem was extending the available SAT solvers to deal directly with atoms in one or more background theories $\mathcal{T}$ instead of encoding them into

CNF.

Procedures for reasoning over background theories ($\mathcal{T} - Solvers$) have been gaining interest since the 1970s and already employed in some domains [47] [54] [20], such as Knowledge Representation and Reasoning, Constraint Satisfaction Problems and AI. However, these procedures could not handle the reasoning with respect to the Boolean component of a formula [51], so procedures which combine theory reasoning with Boolean satisfiability were matured over the last three decades [33] [36] [32] [4] [19] to leverage the strengths of both sides. This combination of procedures are called Satisfiability Modulo Theories or SMT.

Satisfiability Modulo Theories is the problem of deciding the satisfiability of a propositional formula ϕ with respect to a background theory $\mathcal{T}$ [7]. In other words the problem of finding assignment μ in $\mathcal{T}$ that satisfy the formula ϕ. SMT deals with formulas of combination of atoms in propositional logic and others in one or more background theories, as shown in the next example.

$$(x \geq 2) \wedge (x \leq 0 \vee y \geq 0) \wedge A \wedge B \tag{2.16}$$

$x \geq 2$, $x \leq 0$ and $y \geq 0$ are atoms in the theory of linear arithmetic ($\mathcal{T} - atoms$). x and y are integers.

SMT solving is addressed by integrating a SAT solver with one or more theory solvers ($\mathcal{T} - Solvers$). There are two approaches to achieve this integration, the first is to translate the input formula into equi-satisfiable proportional formula then use an off-the-shelf SAT solver to produce a satisfiable assignment. A theory solver ($\mathcal{T} - Solver$) is used to check the validity the assignment with respect to $\mathcal{T}$, this method is called the Eager approach. The second method and the widely adapted by the SMT community is called the Lazy approach [10] [51], in which the $\mathcal{T} - atoms$ in the input formula are abstracted and fed to the SAT solver to produce an assignment. The $\mathcal{T} - Solver$ keeps checking the assignment while it is being built for any conflicts in $\mathcal{T}$ ($\mathcal{T} - Inconsistency$). In case of $\mathcal{T} - Inconsistency$ the $\mathcal{T} - Solver$ generates a conflict clause (lemma) then adds it to the formula and feed it back to the SAT solver. Using the previous formula (2.16) as an example:

$$SMT\ solver\ input:$$

$$(x \geq 2) \wedge (x \leq 0 \vee y \geq 0) \wedge A \wedge B \tag{2.17}$$

$$Abstarcting\ \mathcal{T} - atoms:$$

$$c = (x \geq 2),\ \ d = (x \leq 0)\ and\ e = (y \geq 0) \tag{2.18}$$

$$SAT\ solver\ input:$$

$$c \wedge (d \vee e) \wedge A \wedge B \tag{2.19}$$

$$SAT\ solver\ returns\ a\ partial\ model:$$

$$c = true, d = true \tag{2.20}$$

$$\mathcal{T} - solver\ checks\ the\ partial\ model\ for\ any\ \mathcal{T} - Inconsistency:$$

$$(x \geq 2)\ and\ (x \leq 0)\ are\ in\ Conflict \tag{2.21}$$

$$So\ the\ partial\ model\ (c = true, d = true)\ is\ \mathcal{T} - Inconsistent \tag{2.22}$$

$$\mathcal{T} - solver\ notifies\ the\ SAT\ solver\ about\ the\ conflict \tag{2.23}$$

$$The\ lemma\ \neg(c \wedge d)\ is\ added\ to\ the\ formula \tag{2.24}$$

$$SAT\ solver\ Backjumps\ and\ adjusts\ the\ model \tag{2.25}$$

2.3.1 SMTLIB

SMT-LIB is the standard modelling language for Satisfiability Modulo Theories. It was introduced by the SMT-LIB initiative [7] to serve as a standard benchmarking suite for their annual SMT solvers competition [13] and a standard language and interfaces for the different SMT solvers [8]. The SMT-LIB standard promotes ease of parsing over human readability, that is because SMT-LIB code is meant to be generated by automated modelling tools, also to make it easier for the solvers to parse the code so easier for their developers to adopt the standard [7].

2.3.2 SMT Applications

The modular concept of SMT solvers enables them to easily support new theories which could open the doors for more applications . The scope of applications of SMT solvers currently ranges over software testing and verifications, model checking and theorem proving. Companies like Microsoft developing their own SMT solver [17] and currently using it as a main tool for their software verification and unit test generation [18]. Another well known innovative company, SRI International [2], rely on the integration of their SMT solver Yices to build their theorem proving, model checking and probabilistic consistency tools [22]. Alongside the previous examples, the SMT solver Yices proved to be very competitive on a suit of benchmarks of problems related to scheduling, optimisation, design and others [12].

2.4 Symmetry in Satisfiability

A symmetry in satisfiability can be defined as permutations of a set of assignments of a formula which preserve its satisfiability state. For instance, if M is a set of all assignments that satisfies the formula ϕ, then any bijection $f(M)$ that maintains the satisfiability of ϕ is a symmetry of M. Symmetries arises in models of many satisfiability problems, where pairs of values or variables in a formula can be interchanged without affecting its satisfiability state [40] [50]. For example, because of the property of commutativity $(\neg A \vee B) \wedge (A \vee \neg B)$ has a symmetry between its variables A and B, which maps $A \to B$ and $B \to A$. This type of symmetry is known as variable symmetry. Another type of symmetry appears when the interchangeability is possible between values. For instance, in $(\neg A \vee B) \wedge (A \vee \neg C) \wedge (\neg B \vee C)$ we could flip any assignments of A and B with no effect on the outcome of the formula. Variable and value symmetries could exist simultaneously in the same model, this case is known as the mixed symmetry.

Symmetries in satisfiability solving create redundant paths to solutions as well as to non-solutions in the search space. Eliminating Symmetries could save a solver time and resources spent in exploring those paths in search of a solution [25] [49] [15]. There are two main approaches to symmetry breaking in satisfiability [31] [24], both are based on adding extra constraints to prune the redundant branches of the search tree. The main difference between the two lays on when to add the symmetry breaking constraints. One approach adds a set of symmetry breaking constraints to the problem's model before starting the search, it is called static symmetry breaking. The second, which is known as dynamic symmetry breaking, is based on trying to guide the search process away form symmetries by adding the appropriate symmetry breaking constraints during search.

2.4.1 Symmetry Braking and the Lexicographic Order Constraint

A well known case of symmetries in combinatorial problems is the interchangeability between rows and between columns of matrices of decision variables. An $n \times m$ size matrix has a $n! \times m!$ possible symmetries among each pair of rows and pair columns. The following example shows the possible rows and columns permutations for a 2×3 array.

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix} \begin{pmatrix} b & a & c \\ e & d & f \end{pmatrix} \begin{pmatrix} c & b & a \\ f & e & d \end{pmatrix} \begin{pmatrix} a & c & b \\ d & f & e \end{pmatrix} \begin{pmatrix} c & a & b \\ f & d & e \end{pmatrix} \begin{pmatrix} b & c & a \\ e & f & d \end{pmatrix}$$

$$\begin{pmatrix} d & e & f \\ a & b & c \end{pmatrix} \begin{pmatrix} e & d & f \\ b & a & c \end{pmatrix} \begin{pmatrix} f & e & d \\ c & b & a \end{pmatrix} \begin{pmatrix} d & f & e \\ a & c & b \end{pmatrix} \begin{pmatrix} f & d & e \\ c & a & b \end{pmatrix} \begin{pmatrix} e & f & d \\ b & c & a \end{pmatrix}$$

Row and column symmetries are closely related to some real world problems, specifically whenever matrices are employed to model a problem, which is a common practice in scheduling problems for instance.

The Lexicographic order constraint has been proved useful in breaking certain kinds of symmetries in matrices of decision variables. To break all row and column symmetries Crawford *et al* [15] introduced what is now known as the lex-leader constraints. An example provided by Frisch *et al* [26] shows how to apply row-wise lex-leader on a 2×3 array like the one in the previous example, it is built on the idea that an order of a matrix must be lexicographically less than or equal that all of its permutations, to do so, a matrix is transformed into a single row of elements, starting from Left-Right-Top-Down our previous 2×3 matrix becomes $[a, b, c, d, e, f]$ and the symmetry breaking constraints are presented as follows:

$$\begin{bmatrix} a & b & c & d & e & f \end{bmatrix} \leq_{Lex} \begin{bmatrix} b & a & c & e & d & f \end{bmatrix}$$

$$\begin{bmatrix} a & b & c & d & e & f \end{bmatrix} \leq_{Lex} \begin{bmatrix} c & b & a & f & e & d \end{bmatrix}$$

$$\begin{bmatrix} a & b & c & d & e & f \end{bmatrix} \leq_{Lex} \begin{bmatrix} a & c & b & d & f & e \end{bmatrix}$$

$$\begin{bmatrix} a & b & c & d & e & f \end{bmatrix} \leq_{Lex} \begin{bmatrix} c & a & b & f & d & e \end{bmatrix}$$

$$\begin{bmatrix} a & b & c & d & e & f \end{bmatrix} \leq_{Lex} \begin{bmatrix} b & c & a & e & f & d \end{bmatrix}$$

$$\begin{bmatrix} a & b & c & d & e & f \end{bmatrix} \leq_{Lex} \begin{bmatrix} d & e & f & a & b & c \end{bmatrix}$$

$$\begin{bmatrix} a & b & c & d & e & f \end{bmatrix} \leq_{Lex} \begin{bmatrix} e & d & f & b & a & c \end{bmatrix}$$

$$\begin{bmatrix} a & b & c & d & e & f \end{bmatrix} \leq_{Lex} \begin{bmatrix} f & e & d & c & b & a \end{bmatrix}$$

$$\begin{bmatrix} a & b & c & d & e & f \end{bmatrix} \leq_{Lex} \begin{bmatrix} d & f & e & a & c & b \end{bmatrix}$$

$$\begin{bmatrix} a & b & c & d & e & f \end{bmatrix} \leq_{Lex} \begin{bmatrix} f & d & e & c & a & b \end{bmatrix}$$

$$\begin{bmatrix} a & b & c & d & e & f \end{bmatrix} \leq_{Lex} \begin{bmatrix} e & f & d & b & c & a \end{bmatrix}$$

This will eliminate all the 11 permutations from the previous example retaining only the following assignment:

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

Although Crawford's method is complete, i.e., beaks all row and column symmetries in $n \times m$ matrices, it is impractical specially on large matrices, because it produces $(n! \times m!) - 1$ symmetry breaking constraints, which could add burdens that outweigh the benefit of any potential search space pruning.

Based on Crawford's work and by introducing number of symmetry breaking (SB) constraints linear to number of rows and columns in matrices, Shlyakhter [53] and Flener *et al* [23] independently managed to break not all but a great percentage of symmetries in matrices. This is done by adding a *Lex* constraint between each pair of neighbouring rows and columns, this introduces $(n-1)+(m-1)$ number of SB constraints. Again, using the matrix from the previous examples, this method can be presented as follows:

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

$$\leq_{Lex} \mathit{between\ rows}$$

$$\begin{bmatrix} a & b & c \end{bmatrix} \leq_{Lex} \begin{bmatrix} d & e & f \end{bmatrix}$$

$$\leq_{Lex} \mathit{between\ columns}$$

$$\begin{bmatrix} a & d \end{bmatrix} \leq_{Lex} \begin{bmatrix} b & e \end{bmatrix}$$

$$\begin{bmatrix} b & e \end{bmatrix} \leq_{Lex} \begin{bmatrix} c & f \end{bmatrix}$$

We used this method to model all the symmetry breaking constraints we used in this research.

Lex constraint could be also used to break both variable and values symmetries at the same time [59].

Chapter 3

Encodings for the *Lex* Ordering Constraint

This chapter presents nine different encodings for the *Lex* Ordering Constraint. Each of which is drawn from the constraint solving literature or is a variant of such. Throughout, we consider a non-strict *Lex* constraint between two vectors A and B of finite-domain variables. Both vectors are considered to be of length n. We write such constraint as $A \leq_{lex} B$. We assume that $n \geq 2$, because *Lex* on two vectors of $n = 1$ is simply $A[1] \leq B[1]$.

We shall use the term mzn2smt to refer to Bofill's method of translating a MiniZinc 1.6 [1] specification to SMT 1.2, it is done in two steps, starts by using MiniZinc to produce FlatZinc and then passing this through fzn2smt to produce SMT. We chose this pipeline at this stage of our research just for convenience and we are aware of some of its possible drawbacks. For instance, some constraints could be naturally represented in SMT, but when they go through the translation process they get broken into smaller ones.

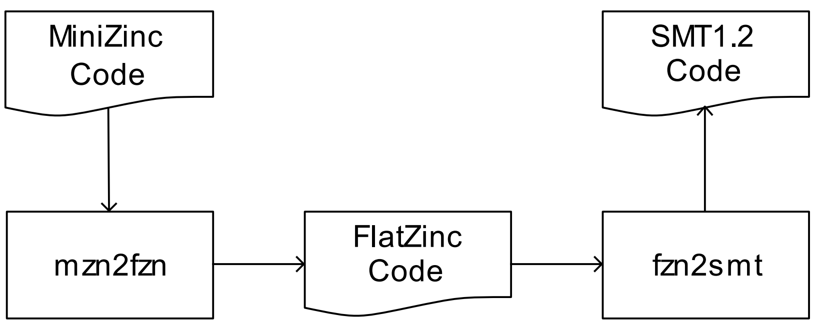

Figure 3.1: The mzn2smt Pipeline

Each of the following subsections presents an encoding of the *Lex* constraint followed by the result of passing it through the mzn2smt pipeline. Since most of the encodings are non-CNF, we decided to measure their sizes by number of atom occurrences, rather than number of constraints . $T_1[i]$, $T_2[i], \ldots$ are auxiliary Boolean arrays introduced by the mzn2smt. The index i of these arrays ranges from 1 and n. The generated SMTLIB code does not literally contain arrays; we use the notation as a clean way of naming a set of n distinct SMT variables.

3.1 The AND Decomposition Encoding

This encoding, which is considered by Frisch *et al* [27], decomposes *Lex* constraint into a conjunction of smaller constraints as shown in the following formula, and because of that it is known as AND Decomposition.

$$
\begin{aligned}
&A[1] \leq B[1] \\
&\bigwedge_{i=1}^{n-1} (\bigwedge_{j=1}^{i} (A[j] = B[j])) \rightarrow (A[i+1] \leq B[i+1])
\end{aligned}
$$

This encoding produces $\frac{(n^2+n)}{2}$ atom occurrences, which makes the produced formula grow quadratically.

A strict ordering can be obtained by adding the constraint $(A[n-1] = B[n-1]) \rightarrow (A[n] < B[n])$ to the encoding and reducing the range of i to $n-2$ in the conjunction.

After translation using mzn2smt:

$$A[1] \leq B[1] \tag{3.1}$$

$$1 \leq i \leq n-1 \qquad T_1[i] \Leftrightarrow (A[i] = B[i]) \tag{3.2}$$

$$1 \leq i \leq n-1 \qquad T_2[i] \Leftrightarrow (A[i+1] \leq B[i+1]) \tag{3.3}$$

$$1 \leq i \leq n-2 \qquad T_3[i] \Leftrightarrow \bigwedge_{j=1}^{i+1} T_1[j] \tag{3.4}$$

$$\neg T_1[1] \vee T_2[1] \tag{3.5}$$

$$1 \leq i \leq n-2 \qquad \neg T_3[i] \vee T_2[i+1] \tag{3.6}$$

This translation also has an $O(n^2)$ growth and produces $\frac{(n^2+13n)}{2}+9$ atom occurrences.

A strict ordering can be obtained by adding the constraint $T_2[n-1] \Leftrightarrow (A[n] < B[n])$ to the encoding and reducing the range of i to $n-2$ in (3.3).

3.2 The AND Decomposition Encoding using Common Sub-expression Elimination

Common Sub-expression Elimination helps in reducing formula size by substituting any recurring parts of the formula with variables. For example, by applying *Lex* using the AND encoding between vectors A and B, both of a size 4, we get the following formula:

$$(A[1] \leq B[1]) \tag{3.7}$$

$$(A[1] = B[1]) \rightarrow (A[2] \leq B[2]) \tag{3.8}$$

$$((A[1] = B[1]) \wedge (A[2] = B[2])) \rightarrow (A[3] \leq B[3]) \tag{3.9}$$

$$((A[1] = B[1]) \wedge (A[2] = B[2]) \wedge (A[3] = B[3])) \rightarrow (A[4] \leq B[4]) \tag{3.10}$$

The above formula has a quadratic size growth, which can be eliminated using the Boolean array $X[i]$ to perform CSE. The following resulting formula has a linear growth:

$$(A[1] \leq B[1]) \tag{3.11}$$

$$X[1] \Leftrightarrow (A[1] = B[1]) \tag{3.12}$$

$$X[2] \Leftrightarrow (X[1] \wedge (A[2] = B[2])) \tag{3.13}$$

$$X[3] \Leftrightarrow (X[2] \wedge (A[3] = B[3])) \tag{3.14}$$

$$X[1] \rightarrow (A[2] \leq B[2]) \tag{3.15}$$

$$X[2] \rightarrow (A[3] \leq B[3]) \tag{3.16}$$

$$X[3] \rightarrow (A[4] \leq B[4]) \tag{3.17}$$

The following encoding, which we call AND CSE, is similar to the AND encoding and produces a similar formula too. The difference is, in this encoding we use a Boolean array to eliminate common sub-expressions in the formula as presented in line (3.20).

The purpose of this encoding is to compare performance between using the nested loops as in line (3.4) in the previous encoding and this approach.

The AND encoding using eliminating common sub-expressions using the Boolean array $X[i]$:

$$A[1] \leq B[1] \tag{3.18}$$

$$X[1] \Leftrightarrow (A[1] = B[1]) \tag{3.19}$$

$$1 \leq i \leq n-2 \qquad X[i+1] \Leftrightarrow (X[i] \wedge (A[i+1] = B[i+1])) \tag{3.20}$$

$$1 \leq i \leq n-1 \qquad X[i] \rightarrow (A[i+1] \leq B[i+1]) \tag{3.21}$$

This formula produces $5n-5$ atom occurrences. A strict ordering can be obtained by adding the constraint $X[n-1] \rightarrow (A[n] < B[n])$ to the encoding and changing the range of i in (3.21) to $n-2$.

After translation using mzn2smt:

$$A[1] \leq B[1] \tag{3.22}$$

$$X[1] \Leftrightarrow (A[1] = B[1]) \tag{3.23}$$

$$1 \leq i \leq n-2 \qquad T_1[i] \Leftrightarrow (A[i+1] = B[i+1]) \tag{3.24}$$

$$1 \leq i \leq n-1 \qquad T_2[i] \Leftrightarrow (A[i+1] \leq B[i+1]) \tag{3.25}$$

$$1 \leq i \leq n-2 \qquad X[i+1] \Leftrightarrow (X[i] \wedge T_1[i]) \tag{3.26}$$

$$1 \leq i \leq n-1 \qquad \neg X[i] \vee T_2[i] \tag{3.27}$$

This formula produces $9n-11$ atom occurrences.

A strict ordering can be obtained by adding the constraint $T_2[n-1] \Leftrightarrow (A[n] < B[n])$ to the encoding and reducing the range of i to $n-2$ in (3.25).

3.3 The OR Decomposition Encoding

This encoding, also considered by Frisch *et al* [27], decomposes the *Lex* constraint into a formula of smaller constraints disjoined together, ie a DNF formula. It is traditionally known as the OR decomposition.

$$(A[1] < B[1]) \vee$$

$$(\bigvee_{i=1}^{n-1}(\bigwedge_{j=1}^{i}(A[j] = B[j])) \wedge (A[i+1] < B[i+1])) \vee \tag{3.28}$$

$$(\bigwedge_{i=1}^{n}(A[i] = B[i])) \tag{3.29}$$

This encoding produces $\frac{(n^2+3n)}{2}$ atom occurrences, which, like AND encoding, makes the produced formula grow quadratically. A strict ordering can be obtained by removing (3.29) from the above formula.

After translation using mzn2smt:

$$1 \leq i \leq n \qquad T_1[i] \Leftrightarrow (A[i] = B[i]) \tag{3.30}$$

$$1 \leq i \leq n \qquad T_2[i] \Leftrightarrow (A[i] < B[i]) \tag{3.31}$$

$$1 \leq i \leq n-1 \qquad T_3[i] \Leftrightarrow \bigwedge_{j=1}^{i} T_1[j] \wedge T_2[i+1] \tag{3.32}$$

$$T_3[n] \Leftrightarrow \bigwedge_{i=1}^{n} T_1[i] \tag{3.33}$$

$$(\bigvee_{i=1}^{n-1} T_3[i]) \vee T_2[1] \vee T_3[n] \tag{3.34}$$

This encoding produces $\frac{(n^2+15n)}{2}$ atom occurrences. A strict ordering can be obtained by removing $T_3[n]$ from (3.34).

3.4 The OR Decomposition Encoding using Common Subexpression Elimination

This OR decomposition, we call it OR CSE, uses a Boolean array to eliminate common sub-expressions from the formula. $X[i]$ is a Boolean array with an index range of 1 to n, it is used to eliminate common sub-expressions as shown in the following formula.

$$
\begin{aligned}
& ((A[1] < B[1]) \vee \\
& \quad (\bigvee_{i=1}^{n-1} X[i] \wedge (A[i+1] < B[i+1])) \ \vee X[n]) && (3.35)\\
& X[1] \Leftrightarrow (A[1] = B[1]) && (3.36)\\
1 \leq i \leq n-1 \qquad & X[i+1] \Leftrightarrow (X[i] \wedge (A[i+1] = B[i+1])) && (3.37)
\end{aligned}
$$

This encoding produces $5n - 1$ atom occurrences. A strict ordering can be obtained by removing $X[n]$ from (3.35).

After translation the OR using mzn2smt we get:

$$
\begin{aligned}
1 \leq i \leq n \qquad & X[1] \Leftrightarrow (A[1] = B[1]) && (3.38)\\
1 \leq i \leq n-1 \qquad & T_2[i] \Leftrightarrow (A[i+1] = B[i+1]) && (3.39)\\
1 \leq i \leq n \qquad & T_3[i] \Leftrightarrow (A[i] < B[i]) && (3.40)\\
1 \leq i \leq n-1 \qquad & X[i+1] \Leftrightarrow (X[i] \wedge T_2[i]) && (3.41)\\
1 \leq i \leq n-1 \qquad & T_4[i] \Leftrightarrow (X[i] \wedge T_3[i+1]) && (3.42)\\
& (\bigvee_{i=1}^{n-1} T_4[i]) \vee T_3[1] \vee X[n] && (3.43)
\end{aligned}
$$

This encoding produces $13n - 7$ atom occurrences. A strict ordering can be obtained by removing $X[n]$ from (3.43).

3.5 The Recursive OR Decomposition

This variant of the OR encoding, presented by Gent *et al* [30], decomposes *Lex* into a set of nested of ORs and ANDs, unwinding them produces the same OR encoding. We produced this encoding using a recursion.

$$A[1] < B[1] \vee (A[1] = B[1] \wedge (A[2] < B[2] \vee (A[2] = B[2] \wedge (... \wedge (A[n] \leq B[n])...))))$$

We get the above encoding using the following constraints. We introduced the boolean array X, of a size n, to eliminate common sub-expressions and to simulate a recursion.

$$
\begin{aligned}
& X[1] && (3.44)\\
& X[n] \Leftrightarrow (A[n] \leq B[n]) && (3.45)\\
1 \leq i \leq n-1 \quad & X[n-i] \Leftrightarrow (A[n-i] < B[n-i] \vee \\
& \quad (A[n-i] = B[n-i] \wedge X[n-i+1])) && (3.46)
\end{aligned}
$$

The Recursive OR Decomposition produces $2n$ atom occurrences. It can be made strict by changing $(A[n] \leq B[n])$ in (3.45) to $(A[n] < B[n])$.

The mzn2smt translation of the recursive OR is as follows:

$$
\begin{aligned}
& X[1] && (3.47)\\
1 \leq i \leq n \quad & T_1[i] \Leftrightarrow (A[1] = B[1]) && (3.48)\\
1 \leq i \leq n \quad & T_2[i] \Leftrightarrow (A[1] < B[1]) && (3.49)\\
& X[n] \Leftrightarrow (T_1[n] \vee T_2[n]) && (3.50)\\
1 \leq i \leq n-1 \quad & T_3[i] \Leftrightarrow (X[i] \Leftrightarrow T_2[i]) && (3.51)\\
1 \leq i \leq n-1 \quad & T_4[i] \Leftrightarrow (T_1[n-i] \Leftrightarrow X[n-i+1]) && (3.52)\\
1 \leq i \leq n-1 \quad & T_4[i] \vee T_3[n-i] && (3.53)
\end{aligned}
$$

This translation produces $12n-4$ atom occurrences. It can be made strict by removing $T_1[n]$ from (3.50).

3.6 The Arithmetic *Lex* Encoding

Another way of encoding *Lex* constraint is using an arithmetic constraint. This constraint considered by Frisch *et al* [27] , it compares the sum of the values of two vectors with each value multiplied by a factor that represents the significance of the values. We assume all the variables in A and B have a domain of 1 *to d*.

$$\sum_{i=1}^{n} A[i] \times d^{n-i} \;\leq\; \sum_{i=1}^{n} B[i] \times d^{n-i}$$

mzn2smt translation produces exactly the same formula above.

This encoding is limited by the size of data type used to represent domains of values, for example, if $A[1] \times d^{n-1}$ exceeds the maximum value that can be stored in a 32-bit integer this would cause an arithmetic overflow and a system error in computers. A strict ordering can be achieved by changing $\leq$ to $<$.

3.7 Harvey *Lex* Encoding

This alternative arithmetic encoding is introduced by Frisch *et al* [27] who attribute it to Warwick Harvey. The general formula of this encoding is:

$$(A[1] < (B[1] + (A[2] < (B[2] + (... + (A[n] < (B[n]) + 1)...)))) = 1$$

To remove the ellipsis and encode the decomposition in Minizinc, we introduce $X[i]$, a Boolean array used to eliminate common sub-expressions, where i is an index with possible values from 1 to $n-1$.

$$X[1] \tag{3.54}$$

$$X[n] \Leftrightarrow (A[n] < (B[n] + 1)) \tag{3.55}$$

$$0 \leq i \leq n-2 \qquad X[n-i-1] \Leftrightarrow (A[n-i-1] < (B[n-i-1] + Bool2Int(X[n-i]))) \tag{3.56}$$

This encoding produces $2n-1$ atom occurrences. We get a strict version by changing $B[n]+1$ to $B[n]+0$ in (3.55). 1 The translation from MiniZinc to SMT using mzn2smt produces the following. $int[i]$ is an integer array introduced by fzn2smt to encode the Bool2Int function of MiniZinc. $int[i]$ has a domain of 0 to 1 and a size of 1 to n

$$X[1] \tag{3.57}$$

$$X[n] \Leftrightarrow ((A[n] - B[n]) \leq 0) \tag{3.58}$$

$$1 \leq i \leq n \qquad int[i] \leq 1 \tag{3.59}$$

$$1 \leq i \leq n \qquad int[i] \geq 0 \tag{3.60}$$

$$1 \leq i \leq n-1 \qquad X[i+1] \rightarrow (int[i] = 1) \tag{3.61}$$

$$1 \leq i \leq n-1 \qquad \neg X[i+1] \rightarrow (int[i] = 0) \tag{3.62}$$

$$1 \leq i \leq n-1 \qquad X[n-i] \Leftrightarrow ((A[n-i] - B[n-i] - int[i]) \leq -1) \tag{3.63}$$

This translation produces $8n - 3$ number of atom occurrences. We get a strict version by changing $((A[n] - B[n]) \leq 0)$ to $((A[n] - B[n]) < 0)$ in (3.58).

3.8 Alpha *Lex* Encoding

This encoding was introduced by Gent *et al* [30], we called it Alpha, because it uses a Boolean array as an index to track the relations between values. This Boolean array is called $\alpha[i]$ and behaves as follows:

$$1 \leq i \leq n$$

$$1 \leq j \leq i$$

$$if\ \alpha[i] = true\ then\ A[j] = B[j] \tag{3.64}$$

$$if\ (\alpha[i] = true\ and\ \alpha[i+1] = false)\ then\ A[i+1] < B[i+1] \tag{3.65}$$

This makes all values from $\alpha[1]$ to $\alpha[i]$ equal to 1 while $A[i] = B[i]$ holds, and equal to 0 from the first occurrence of $A[i] < B[i]$ till the end of vectors.

$$\alpha[0] \tag{3.66}$$

$$0 \le i \le n-1 \qquad \neg\alpha[i] \rightarrow \neg\alpha[i+1] \tag{3.67}$$

$$1 \le i \le n \qquad \alpha[i] \rightarrow (A[i] = B[i]) \tag{3.68}$$

$$0 \le i \le n-1 \qquad ((\alpha[i]) \wedge (\neg\alpha[i+1])) \rightarrow (A[i+1] < B[i+1]) \tag{3.69}$$

$$0 \le i \le n-1 \qquad \alpha[i] \rightarrow (A[i+1] \le B[i+1]) \tag{3.70}$$

This encoding behaves in a way similar to the AND encoding. Line (3.66) is to guarantee that $A[1] \le B[1]$ and if $A[1] = B[1]$ then $\alpha[1] = true$, which in turn, implies that the next index of A is less than or equal the next index of B, this goes on till the end of the vectors. The difference between the two encodings appears with the first occurrence of $A[i] < B[i]$, where ALPHA uses the constraints in lines (3.67) and (3.69) to make the values of $A[i]$ and $B[i]$ after the first occurrence of $A[i] < B[i]$ insignificant to the problem.

This encoding can be changed to a strict *Lex* by adding the constraint $\neg\alpha[n+1]$ to the formula. The encoding produces $9n-6$ atom occurrences.

Next is the encoding's mzn2smt translation. Here we use $\alpha^{'}$ instead of α because the mzn2smt translator changed the range of α from $0 \le i \le n$ to $1 \le i \le n+1$

$$\alpha^{'}[1] \tag{3.71}$$

$$1 \le i \le n+1 \qquad T_1[i] \Leftrightarrow \neg\alpha^{'}[i+1] \tag{3.72}$$

$$1 \le i \le n \qquad T_2[i] \Leftrightarrow (A[i] = B[i]) \tag{3.73}$$

$$1 \le i \le n \qquad T_3[i] \Leftrightarrow (A[i] < B[i]) \tag{3.74}$$

$$1 \le i \le n \qquad T_4[i] \Leftrightarrow (A[i] \le B[i]) \tag{3.75}$$

$$1 \le i \le n \qquad T_5[i] \Leftrightarrow (\alpha[i] \wedge T_1[i+1]) \tag{3.76}$$

$$1 \le i \le n \qquad \neg T_1[i] \vee T_1[i+1] \tag{3.77}$$

$$1 \le i \le n \qquad \neg T_5[i] \vee T_3[i] \tag{3.78}$$

$$1 \le i \le n \qquad \neg\alpha^{'}[i] \vee T_4[i] \tag{3.79}$$

$$1 \le i \le n \qquad \neg\alpha^{'}[i+1] \vee T_2[i] \tag{3.80}$$

This translation produces $18n-3$ atom occurrences and can be changed to a strict *Lex* by adding the constraint $\neg\alpha[n+1]$

3.9 Alpha M *Lex* Encoding

This decomposition, which we call Alpha M, is the default decomposition used by the Minizinc 1.6 [1]. Like the previous Alpha encoding it uses a Boolean array as a bookkeeping mechanism for relations between the corresponding values in both vectors. The index of the Alpha array ranges from 1 to $n+1$.

$$\alpha[1] \tag{3.81}$$

$$1 \leq i \leq n \qquad \alpha[i] \Leftrightarrow (((A[i] < B[i]) \vee \alpha[i+1]) \wedge (A[i] \leq B[i])) \tag{3.82}$$

Like in AND and ALPHA encodings, ALPHA M makes sure that $A[1] \leq B[1]$ by setting $\alpha[1]$ to $true$, which in line (3.82) guarantees that $A[2] \leq B[2]$ and $\alpha[2] = true$, this continues with each next $A[i]$, $B[i]$ and $\alpha[i]$ till the first occurrence of $A[i] < B[i]$, where afterwards $\alpha[i+1]$ becomes $false$ and all values of $A[i+1]$, $B[i+1]$ and $\alpha[i+2]$ become insignificant.

This encoding produces $4n+1$ number of atom occurrences and we can obtain a strict version by adding $\neg\alpha[n+1]$ to the constraints.

After translation using mzn2smt :

$$\alpha[1] \tag{3.83}$$

$$1 \leq i \leq n \qquad T_1[i] \Leftrightarrow (A[i] \leq B[i]) \tag{3.84}$$

$$1 \leq i \leq n \qquad T_2[i] \Leftrightarrow (A[i] < B[i]) \tag{3.85}$$

$$1 \leq i \leq n \qquad T_3[i] \Leftrightarrow (T_2[i] \vee \alpha[i+1]) \tag{3.86}$$

$$1 \leq i \leq n \qquad \alpha[i] \Leftrightarrow (T_3[i] \wedge T_1[i]) \tag{3.87}$$

This translation produces $10n+1$ number of atom occurrences and could be changed to a strict *Lex* by also adding the constraint $\neg\alpha[n+1]$

Chapter 4

Experimental Results

We evaluated eight of the decompositions on solving a suite of instances of the Social Golfers Problem (SGP) and Balanced Incomplete Block Designs Problem (BIBD), problems 010 and 028 in CSPLib [38]. Both problems are well known for their highly symmetric models. The arithmetic decomposition is not evaluated because it is impractical to do so. We also run inference tests on a set of unsatisfiable instances of enforcing *Lex* between two long vectors. To chose the benchmarking SMT solvers, we ran performance comparison tests between four SMT solvers, Yices1, Yices2 [22], Z3 [17] and CVC4 [9]. We decided to use the two with the best performances and the best output format, these were Yices2 and Z3.

We ran two sets of benchmarks to evaluate the decompositions. The first, is by directly translating each encoding to SMT using the C# programming language then benchmark them. In the second, we were aiming to test the effect of using constraint reification on the SMT decompositions of *Lex*. Constraint reification was the method chosen by Bofill *et al* to translate CSP problems into SMT [12], which alongside the SMT solver Yices2, proved competitive against some leading CSP solvers. Reifying a constraint C is reformulating it to the form of $(b \Leftrightarrow C)$, where b is a boolean proposition. We used two off-the-shelf tools, Minizinc's mzn2fzn [1] and Bofill's fzn2smt to get the reified constraints translations for the decompositions.

Both sets of benchmarks share the same code for the benchmarking problems, the difference is only in the code related to the different *Lex* constraint encodings.

We made the direct translation using a C# language code. As for the mzn2smt translation, the MiniZinc implementation includes a set of libraries to decompose global constraints and made to be called from MiniZinc models. We created a similar MiniZinc global library for each of the eight *Lex* decompositions, then we called them from the benchmarking problem's code. We modified a MiniZinc model for the problems by adding a symmetry breaking based on lexicographical orderings constraint.

Frisch *et al* [27] proved that the conjunction of AND and OR encodings facilitates a stronger Generalised Arc Consistency (GAC) than each separately, so we decided to include AND $\wedge$ OR in the Long Vectors instances tests and call it ANDOR. We did not do ANDOR tests on the SGP and the BIBD because the size of the ANDOR caused Out-of-Memory problems on most of the instances.

All benchmarks were run on a Windows PC with Intel i7 1.8Ghz processor and 8GB of RAM and using Yices2 v2.2.1 and Z3 v4.3.0 SMT solvers on the SGP, the BIBD problems and on the unsatisfiable instances.

We noticed that different orders of an SMT file have different solving times. To get a mean value of solving times, we decided to run 30 samples for each instance. Each sample is created by choosing a random ordering of the constraints from a uniform distribution over all orderings of the sample SMT file. To generate the 30 samples we made a shuffling script. Each figure in the tables from table 4.1 to 4.24 represents an average of 30 values for each encoding on each instance. Figure 4.1 demonstrates the process of obtaining the samples. We set a time-out of 300 seconds for each run for both the SMT solver.

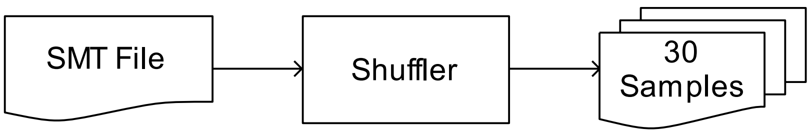

Figure 4.1: Generating the Samples

Apart from the instance 8-4-4 of the SGP, we only used satisfiable instances for both the SGP and the BIBD. Instances of the SGP were obtained from [35], as for the BIBD we made our own. The instances were chosen to represent different levels of difficulty.

4.1 The Social Golfers Problem

The Social Golfers Problem is a computational problem of partitioning a set of golfers into g groups of size s in each of w weeks such that no two players meet more that once in the same group. An instance of the Social Golfer problem is usually denoted $g - s - w$, which stand for number of groups, the group size and number of weeks. We use $m = g \times s$ to denote the number of players

The table below shows one possible solution to the instance 3-2-3, where rows and columns represent players and weeks respectively, and each value in the table denotes a group number. So as an example column 2 can be interpreted as follows; In $Week_2$, $Player_1$ and $Player_3$ meet in the first group, $Player_2$ and $Player_5$ meet in the second group and $Player_4$ and $Player_6$ meet in the third.

$Week_1$	$Week_2$	$Week_3$	
1	1	1	$Player_1$
1	2	2	$Player_2$
2	1	2	$Player_3$
2	3	3	$Player_4$
3	2	3	$Player_5$
3	3	1	$Player_6$

The Social Golfer is known for its highly symmetric models. For example, in the previous solution of the instance 3-2-3 of the SGP, we could swap any two rows or two columns and still get a valid solution. The solution below is a result of swapping the first and last rows.

$Week_1$	$Week_2$	$Week_3$	
3	3	1	$Player_1$
1	2	2	$Player_2$
2	1	2	$Player_3$
2	3	3	$Player_4$
3	2	3	$Player_5$
1	1	1	$Player_6$

We use the *Lex* constraint to break two groups of symmetries in the problem; symmetries in weeks (columns) and symmetries in players (rows). The MiniZinc model that we used for the problem maps Players and Weeks to groups in an array as above. Symmetry among the players is broken by constraining the rows to be in *Lex* increasing order and symmetry among the weeks is broken by constraining the columns to be in *Lex* increasing order.

The model that we used for the SGP is a modified model created by H. Kjellerstrand [41] and it has two constraints : The first is to make all groups contain s players, while the second is to make sure that each two players play together at most once in each week.

$Schedule[,]$ is a two dimensional integer array that holds the weekly assignment of players to groups. Each group has exactly s players:

$$1 \leq group \leq g \quad 1 \leq week \leq w \quad (\sum_{player=1}^{m} Bool2Int(Schedule[player, week] = group)) = s$$

Where Bool2Int() is Boolean to integer converter function.

Each pair of players meet at most once

$$
\begin{aligned}
&1 \leq pa \leq m \quad 1 \leq pb \leq m \\
&1 \leq wa \leq w \quad 1 \leq wb \leq w \\
&where\ pa \neq pb \wedge wa \neq wb \\
&(Schedule[pa, wa] \neq Schedule[pb, wa]) \vee \\
&(Schedule[pa, wb] \neq Schedule[pb, wb])
\end{aligned}
$$

For any two distinct players, pa and pb, and any two distinct weeks, wa and wb, players pa and pb cannot play in the same group in both week wa and wb.

From the assignment array $Schedule[,]$ it is clear that symmetries can happen between weeks and between players. To break symmetry between weeks we put *lex* constraint ordering between each two neighbouring columns and the same is done for players.

Lex constraint on weeks:

$$
\begin{aligned}
1 \leq week \leq w-1 \quad & [Schedule[player, week] \mid player \in 1..m] \leq_{lex} \\
& [Schedule[player, week+1] \mid player \in 1..m]
\end{aligned}
$$

Lex constraint on players:

$$
\begin{aligned}
1 \leq player \leq m-1 \quad & [Schedule[player, week] \mid week \in 1..w] \leq_{lex} \\
& [Schedule[player+1, week] \mid week \in 1..w]
\end{aligned}
$$

4.1.1 SGP Results using Yices2

The results of the SGP using Yices2 are arranged in four tables, two for timings for the directly translated instances and their corresponding numbers of decisions, and similar two for the mzn2smt translation. From Tables 4.1 and 4.4, apart from the Recursive OR (ROR) and Harvey's, all other encoding show similar performance with very minor differences between timings. ROR's average timing is greatly reduced by its timeing-out on the instance 5-3-7, and a smaller effect by the instances 6-3-5 and 5-3-6, otherwise it performed better than all of the other instances, and on most of the larger instances.

Also, from the two tables, it is clear that using constraint reification greatly improved ROR and Harvey's timings, with not much effect on the others. Also, CSE made a negligible difference in cases of the AND and the OR. Tables 4.3 and 4.6 show that, in general, ROR is much faster in making decisions than all the others on most of the instances. Decisions are assigning values to variables, and they depend on the solver's ability to make certain inferences. Those faster decisions imply that the ROR encoding facilitates solving the instances using faster or fewer inferences compared to the other encodings. We compute decision rates by dividing the average number of decisions by the average solution time for each encoding on each instance.

Instances G-S-W	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey	No *Lex*
5-3-5	0.27	0.28	0.27	0.29	**0.17**	0.27	0.25	0.42	0.38
5-3-6	1.93	1.88	2.07	**1.53**	9.66	1.91	1.78	3.13	164.85
5-3-7	12.36	13.59	13.65	16.18	300.00	**10.73**	17.84	32.51	265.53
6-3-5	0.26	0.27	0.29	0.34	0.17	0.27	0.26	0.48	**0.15**
6-3-6	1.54	1.76	1.94	1.88	**0.97**	1.69	1.83	3.18	1.69
6-3-7	7.27	7.26	7.24	6.98	12.50	6.94	**6.93**	10.56	119.99
6-4-4	0.72	0.69	**0.65**	0.66	**0.42**	0.65	0.68	1.11	1.15
6-4-5	4.81	5.06	5.04	4.72	**2.96**	4.74	4.64	6.34	9.25
8-4-4	1.12	1.03	1.21	1.25	**0.62**	1.16	1.00	1.71	0.88
8-4-5	10.41	10.57	9.89	10.78	**6.68**	10.08	10.56	21.26	23.56
8-4-6	82.84	82.16	81.60	78.15	**36.75**	90.06	82.68	127.09	57.63
Arith-mean	11.23	11.32	11.26	**11.16**	33.72	11.68	11.68	18.89	58.63
Geo-mean	2.73	2.78	2.84	2.86	3.01	**2.71**	2.76	4.66	7.80

Table 4.1: Average solution time (in seconds) for instances of the SGP using the direct translation and Yices2 SMT solver

Instances G-S-W	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
5-3-5	8	10	8	10	12	8	8	13
5-3-6	41	39	41	34	79	34	35	50
5-3-7	135	114	99	133	2,163	92	141	192
6-3-5	10	12	11	12	16	9	7	18
6-3-6	50	51	54	57	62	54	56	82
6-3-7	163	158	162	164	296	177	152	199
6-4-4	22	23	20	19	29	22	20	31
6-4-5	122	127	128	134	136	125	114	132
8-4-4	66	62	72	65	98	61	59	98
8-4-5	467	405	347	449	607	429	409	727
8-4-6	2,154	2,446	2,443	2,311	2,135	2,552	2,209	3,051

Table 4.2: Average number of decisions (divided by 1000) for instances of the SGP using the direct translation and Yices2 SMT solver

Instances G-S-W	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
535	30	35	31	35	**68**	29	34	31
536	21	21	20	**22**	8	18	20	16
537	**11**	8	7	8	7	9	8	6
635	39	43	39	36	**99**	32	27	37
636	32	29	28	30	**64**	32	31	26
637	22	22	22	23	24	**26**	22	19
644	30	34	31	28	**69**	34	30	28
645	25	25	25	28	**46**	26	25	21
844	59	60	59	52	**158**	53	59	57
845	45	38	35	42	**91**	43	39	34
846	26	30	30	30	**58**	28	27	24

Table 4.3: Average number of decisions per millisecond for instances of the SGP using the direct translation and Yices2 SMT solver

Instances G-S-W	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey	No *Lex*
5-3-5	0.27	**0.25**	0.28	0.26	**0.17**	0.25	0.27	0.33	0.38
5-3-6	1.40	1.79	1.70	2.02	2.46	1.60	**1.59**	1.94	164.85
5-3-7	12.53	13.08	**11.07**	13.94	265.0	13.47	14.77	16.71	265.85
6-3-5	0.27	0.27	0.25	0.26	0.16	0.26	0.29	0.32	**0.15**
6-3-6	1.63	1.64	1.70	1.79	**1.03**	1.69	1.55	1.87	1.69
6-3-7	6.77	6.72	7.14	7.32	28.92	**6.10**	6.84	6.81	119.99
6-4-4	0.63	0.63	0.62	0.63	**0.47**	0.66	0.64	0.68	1.15
6-4-5	5.32	4.82	5.01	4.53	**3.14**	4.97	4.88	4.89	9.25
8-4-4	1.08	0.98	1.11	1.17	**0.67**	1.09	1.14	1.20	0.88
8-4-5	10.88	10.80	10.42	9.73	**6.77**	10.76	9.63	10.80	23.56
8-4-6	85.45	87.43	79.19	90.63	**41.51**	84.18	93.65	83.29	57.49
Arith-mean	11.48	11.67	**10.77**	12.03	31.85	11.37	12.29	11.71	58.63
Geo-mean	2.66	2.67	**2.65**	2.77	2.95	2.66	2.73	2.97	7.80

Table 4.4: Average solution time (in seconds) for instances of the SGP using mzn2smt translation and Yices2 SMT solver

Instances G-S-W	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
5-3-5	10	10	10	9	14	8	9	11
5-3-6	32	34	37	38	48	35	34	40
5-3-7	129	113	94	123	629	115	161	128
6-3-5	11	10	10	11	16	10	11	13
6-3-6	52	59	63	56	63	59	55	61
6-3-7	150	159	174	172	181	128	162	169
6-4-4	22	23	24	22	31	22	23	21
6-4-5	139	106	121	131	136	129	127	125
8-4-4	67	61	61	69	109	62	71	67
8-4-5	431	407	420	375	652	446	425	472
8-4-6	2,235	2,507	2,155	2,521	2,645	2,290	2,728	2,483

Table 4.5: Average number of decisions (divided by 1000) for instances of the SGP using mzn2smt translation and Yices2 SMT solver

Instances G-S-W	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
535	37	38	35	34	**82**	31	33	35
536	20	18	21	20	19	**22**	21	19
537	**12**	9	9	10	2	9	**12**	8
635	41	37	39	42	**100**	40	39	39
636	32	36	39	32	**61**	36	32	34
637	21	24	**25**	24	6	20	23	23
644	35	36	38	36	**67**	36	34	30
645	27	22	25	29	**43**	26	26	26
844	64	64	56	59	**163**	58	61	56
845	41	42	40	38	**96**	42	42	43
846	25	29	28	29	**64**	26	28	30

Table 4.6: Average number of decisions per millisecond for instances of the SGP using mzn2smt translation and Yices2 SMT solver

4.1.2 SGP Results using Z3

Similar to SGP using Yices2, apart from the ROR and Harvey encodings, the results here show similar performances of Z3 on all instances. The dominance of the ROR encoding is much evident here. Z3 results also supports that ROR facilitates a faster decision rate. CSE did not make a noticeable difference in cases of the AND and the OR. Constraint reification slightly helped Harvey's, but had contrary small effect on the others.

Instances G-S-W	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey	No *Lex*
5-3-5	1.17	1.14	1.17	1.19	0.44	1.18	1.14	1.44	**0.27**
5-3-6	6.80	7.26	7.63	8.19	**3.05**	7.67	8.06	18.17	5.26
5-3-7	44.86	51.42	44.17	56.74	**34.52**	48.16	42.15	98.45	173.12
6-3-5	2.30	2.24	2.43	2.35	0.61	2.38	2.33	2.66	**0.38**
6-3-6	4.37	4.33	4.62	4.72	1.03	4.48	4.48	5.39	**0.70**
6-3-7	15.67	15.70	19.15	19.59	**4.77**	17.98	15.86	25.36	5.31
6-4-4	3.40	3.45	3.79	3.68	0.83	3.53	3.47	4.03	**0.57**
6-4-5	9.53	10.29	10.09	10.45	**2.29**	9.56	9.67	15.69	2.55
8-4-4	10.94	11.18	12.74	12.77	1.99	11.40	11.18	12.71	**1.20**
8-4-5	31.64	31.82	38.22	38.04	3.97	30.58	32.17	38.81	**2.77**
8-4-6	59.38	65.06	58.76	50.47	9.92	69.05	66.38	42.88	**8.24**
Arith-Mean	17.28	18.54	18.43	18.93	**5.76**	18.72	17.90	24.14	18.22
Geo-Mean	9.14	9.45	9.93	10.11	2.47	9.65	9.41	12.59	**2.38**

Table 4.7: Average solution time (in seconds) for instances of the SGP using the direct translation and Z3 SMT solver

Instances G-S-W	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
5-3-5	1.36	1.31	1.70	1.83	1.32	1.32	1.38	2.15
5-3-6	13.33	14.39	16.26	17.22	13.15	14.99	16.55	37.22
5-3-7	113.56	132.28	114.73	148.88	176.81	117.69	107.17	196.62
6-3-5	2.13	2.01	2.84	2.36	1.90	1.89	2.22	3.14
6-3-6	3.53	3.34	4.65	4.77	3.51	3.47	3.71	5.64
6-3-7	22.68	22.65	34.05	32.32	17.83	26.35	22.98	47.79
6-4-4	2.29	2.30	3.66	3.22	2.57	2.32	2.43	3.78
6-4-5	7.47	8.31	9.60	10.38	6.24	7.53	7.95	17.98
8-4-4	4.94	5.01	8.64	8.80	5.77	4.86	5.08	8.78
8-4-5	18.96	20.52	35.14	34.69	11.70	17.41	21.56	35.85
8-4-6	56.55	49.70	64.47	68.98	37.27	47.00	49.42	80.34

Table 4.8: Average number of decisions (divided by 1000) for instances of the SGP using the direct translation and Z3 SMT solver

Instances G-S-W	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
535	1	1	1	2	**3**	1	1	1
536	2	2	2	2	**4**	2	2	2
537	3	3	3	3	**5**	2	3	2
635	1	1	1	1	**3**	1	1	1
636	1	1	1	1	**3**	1	1	1
637	1	1	2	2	**4**	1	1	2
644	1	1	1	1	**3**	1	1	1
645	1	1	1	1	**3**	1	1	1
844	1	1	1	1	**3**	1	1	1
845	1	1	1	1	**3**	1	1	1
846	1	1	1	1	**4**	1	1	2

Table 4.9: Average number of decisions per millisecond for instances of the SGP using the direct translation and Z3 SMT solver

Instances G-S-W	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey	No *Lex*
5-3-5	1.16	1.19	1.28	1.27	0.43	1.26	1.17	1.35	**0.27**
5-3-6	8.40	7.62	7.63	8.23	**3.56**	10.13	7.70	10.41	5.26
5-3-7	44.20	42.19	53.84	54.60	**29.89**	41.99	49.93	53.69	173.12
6-3-5	2.31	2.28	2.51	2.44	0.60	2.43	2.36	2.57	**0.38**
6-3-6	4.58	4.51	4.92	4.86	1.05	4.60	4.52	5.02	**0.70**
6-3-7	16.46	16.50	20.12	18.05	**5.08**	17.03	15.88	20.16	5.31
6-4-4	3.37	3.41	3.71	3.78	0.87	3.55	3.40	3.71	**0.57**
6-4-5	9.77	9.74	11.29	11.65	**2.43**	10.73	9.81	10.39	2.55
8-4-4	11.02	11.02	12.65	13.49	2.10	12.10	11.85	12.16	**1.20**
8-4-5	31.81	31.46	39.30	39.33	4.04	31.11	31.61	34.75	**2.77**
8-4-6	65.38	69.76	46.14	52.66	10.68	66.54	67.83	57.28	**8.24**
Arith-Mean	18.04	18.15	18.49	19.12	**5.52**	18.32	18.73	19.23	18.22
Geo-Mean	9.50	9.43	10.21	10.38	2.53	9.97	9.61	10.56	**2.38**

Table 4.10: Average solution time (in seconds) for instances of the SGP using the mzn2smt translation and Z3 SMT solver

Instances G-S-W	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
5-3-5	1.23	1.27	1.95	1.86	1.24	1.38	1.29	1.83
5-3-6	16.67	14.83	15.80	17.37	15.54	20.11	14.90	20.86
5-3-7	100.71	97.57	133.10	139.69	145.75	96.25	119.15	125.43
6-3-5	1.99	2.06	3.09	2.81	1.77	1.96	2.14	2.60
6-3-6	3.38	3.56	5.24	4.97	3.45	3.48	3.57	4.48
6-3-7	23.81	24.28	36.47	30.91	19.56	25.99	24.02	35.19
6-4-4	2.19	2.16	3.47	3.59	2.40	2.50	2.33	3.15
6-4-5	7.94	7.59	11.89	12.56	6.28	8.53	7.78	9.13
8-4-4	4.93	4.87	8.98	10.90	5.54	5.26	5.21	6.94
8-4-5	20.25	18.83	35.88	35.83	11.07	17.15	17.80	24.99
8-4-6	50.73	47.27	73.33	67.63	38.04	49.24	48.99	63.10

Table 4.11: Average number of decisions (divided by 1000) for instances of the SGP using the mzn2smt translation and Z3 SMT solver

Instances G-S-W	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
535	1	1	2	1	**3**	1	1	1
536	2	2	2	2	**4**	2	2	2
537	2	2	2	3	**5**	2	2	2
635	1	1	1	1	**3**	1	1	1
636	1	1	1	1	**3**	1	1	1
637	1	1	2	2	**4**	2	2	2
644	1	1	1	1	**3**	1	1	1
645	1	1	1	1	**3**	1	1	1
844	1	1	1	1	**3**	1	1	1
845	1	1	1	1	**3**	1	1	1
846	1	1	2	1	**4**	1	1	1

Table 4.12: Average number of decisions per millisecond for instances of the SGP using mzn2smt translation and Z3 SMT solver

4.2 The Balanced Incomplete Block Design

The Balanced Incomplete Block Design is a classic combinatorial problem and it has some applications in design theory [5] [57]. The BIBD is the problem of finding a design of v distinct objects into b blocks in which each block has exactly k distinct objects, every object appears in r blocks and each two distinct objects appear together in λ blocks [48]. An instance of the BIBD is donated (v, b, k, r, λ).

The BIBD MiniZinc model that we used is a part of the MiniZinc benchmark suite [1]. We modified the model by removing its symmetry breaking constraints and adding ones based on the *Lex* ordering constraint. The model uses a 0/1 $[v, b]$ matrix to hold the block designs. One possible solution for the instance (7,7,4,4,2) could be represented by the following matrix, where columns are *Blocks* and rows are *Objects*. A value of 1 represents the occurrence of an $Object_v$ in a $Block_b$ while 0 represents its absence.

$Block_1$	$Block_2$	$Block_3$	$Block_4$	$Block_5$	$Block_6$	$Block_7$	
0	1	1	0	0	1	1	$Object_1$
0	1	0	1	1	0	1	$Object_2$
1	0	1	0	1	0	1	$Object_3$
0	0	1	1	1	1	0	$Object_4$
1	0	0	1	0	1	1	$Object_5$
1	1	0	0	1	1	0	$Object_6$
1	1	1	1	0	0	0	$Object_7$

The model implements the BIBD by using three constraints that represent the three BIBD rules mentioned earlier. Assuming $M[,]$ is the BIBD matrix, since $M[,]$ is a 0/1 matrix the BIBD rule of each $Block_b$ has exactly k distinct *Objects* could be modelled as every column must sum to k.

$$1 \leq Block \leq b \quad (\sum_{Object=1}^{v} M[Object, Block]) = k$$

Likewise, the second rule which states that every object must appear in r blocks becomes every row must sum to r.

$$1 \leq Object \leq v \quad (\sum_{Block=1}^{b} M[Object, Block]) = r$$

The last rule, which restricts the number of *Blocks* that each two *Objects* could appear together to λ is modelled as the dot product of every pair of distinct rows must equal to λ.

$$1 \leq Oa < Ob \leq v \quad (\sum_{Block=1}^{b} (M[Oa, Block] \times M[Ob, Block])) = \lambda$$

It is also worth mentioning that the values of both b and v can be obtained from k, r and λ using the formulas:

$$b = \frac{\lambda \times v \times (v-1)}{k \times (k-1)}$$

$$r = \frac{\lambda \times (v-1)}{k-1}$$

To break row and column symmetries in the BIBD model we used two *Lex* constraints , one on each two neighbouring rows (*Objects*) and similar one for columns (*Blocks*).

Lex constraint on *Object*:

$$\begin{aligned} 1 \leq Object \leq v-1 \quad & [M[Object, Block] \mid Block \in 1..b] \leq_{lex} \\ & [M[Object+1, Block] \mid Block \in 1..b] \end{aligned}$$

Lex constraint on *Blocks*:

$$\begin{aligned} 1 \leq Blocks \leq b-1 \quad & [M[Object, Block] \mid Object \in 1..v] \leq_{lex} \\ & [M[Object, Block+1] \mid Object \in 1..v] \end{aligned}$$

4.2.1 BIBD Results using Yices2

Similar to the SGP, the results of the BIBD using Yices2 are arranged in four tables, two for timings for the directly translated instances and their corresponding numbers of decisions, and likewise two for the mzn2smt translation. Here the difference in performances are much apparent. From Tables 4.13 and 4.16, ROR's lead is much evident in both. Constraint reification helped in some cases of Harvey and OR, but its effect was opposite in cases of Alpha and OR CSE, it also seems that using constraint reification along side CSE reduced the performance of both AND and OR. Like in SGP results, here too, the ROR encoding in general had fastest decision rates.

Instances r, v, λ	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey	No *Lex*
7-3-2	0.11	0.13	0.26	0.24	0.11	0.12	0.10	0.18	**0.08**
8-4-3	0.39	0.30	0.84	0.80	0.31	0.28	0.29	0.69	**0.16**
8-4-4	0.28	0.38	0.41	0.34	**0.07**	0.23	0.11	299	300
9-3-1	0.22	0.19	0.45	0.44	0.10	0.18	0.18	0.29	**0.04**
11-5-2	1.29	1.27	10.40	4.59	1.48	1.04	1.08	1.91	**0.34**
13-3-1	17.1	14.2	31.7	35.6	11.9	16.5	20.3	24.5	**9.96**
13-4-1	3.13	3.53	6.86	6.72	2.63	3.88	4.97	2.25	**0.29**
Arith-mean	**2.23**	2.87	7.28	6.96	2.38	3.18	3.86	47.12	44.41
Geo-mean	0.79	0.78	1.90	1.64	**0.57**	0.71	0.67	2.73	0.76

Table 4.13: Average solution time (in seconds) for instances of the BIBD using the direct translation and Yices2 SMT solver

Instances r, v, λ	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
7-3-2	1,481	1,453	3,380	3,387	2,764	2,201	2,939	2,693
8-4-3	2,356	2,410	6,051	5,927	4,071	2,555	2,704	5,314
8-4-4	1,087	1,232	2,310	2,203	577	956	289	233,823
9-3-1	1,532	1,639	4,641	4,330	2,822	2,016	1,987	3,293
11-5-2	2,847	2,717	9,123	8,861	6,421	3,124	4,172	6,472
13-3-1	25,950	25,316	84,718	81,233	44,060	30,539	33,849	68,440
13-4-1	4,224	4,446	13,444	13,213	8,427	6,021	7,401	10,024

Table 4.14: Average number of decisions for instances of the BIBD using the direct translation and Yices2 SMT solver

Instances r, v, λ	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
7-3-2	13	11	13	14	**25**	18	29	15
8-4-3	6	8	7	7	**13**	9	9	8
8-4-4	4	3	6	6	**8**	4	3	1
9-3-1	7	9	10	10	**16**	11	11	11
11-5-2	2	2	1	2	**4**	3	4	3
13-3-1	2	2	3	2	**4**	2	2	3
13-4-1	1	1	2	2	**3**	2	1	4

Table 4.15: Average number of decisions per millisecond for instances of the BIBD using the direct translation and Yices2 SMT solver

Instances r, v, λ	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey	No *Lex*
7-3-2	0.11	0.11	0.27	0.25	0.10	0.16	0.19	0.19	**0.08**
8-4-3	0.34	**0.30**	0.55	0.76	0.30	0.29	0.53	0.38	**0.16**
8-4-4	0.26	0.31	0.46	0.37	**0.08**	0.34	1.03	0.22	300
9-3-1	0.21	0.20	0.40	0.48	0.16	0.23	0.33	0.30	**0.04**
11-5-2	1.12	1.38	3.72	3.73	1.28	1.33	1.51	7.52	**0.34**
13-3-1	13.23	16.90	28.60	32.77	10.30	56.87	19.09	23.72	**9.96**
13-4-1	4.07	4.52	9.45	21.69	2.82	3.05	3.38	3.98	**0.29**
Arith-mean	2.76	3.39	6.21	8.58	**2.15**	8.89	3.72	5.18	44.41
Geo-mean	0.74	0.80	1.60	1.91	**0.55**	0.97	1.19	1.18	0.76

Table 4.16: Average solution time (in seconds) for instances of the BIBD using mzn2smt translation and Yices2 SMT solver

Instances r, v, λ	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
7-3-2	1,456	1,450	3,784	3,812	5,507	1,962	1,914	2,493
8-4-3	2,407	2,182	5,077	5,806	8,235	2,582	3,398	2,983
8-4-4	864	1,066	2,335	2,320	682	1,036	2,788	436
9-3-1	1,573	1,520	4,007	4,571	6,495	2,074	2,296	3,107
11-5-2	2,590	2,926	8,416	9,044	15,607	3,168	4,203	5,869
13-3-1	24,013	26,956	70,828	83,951	91,073	29,703	35,521	44,136
13-4-1	4,398	4,960	14,919	25,726	22,862	5,111	5,654	11,060

Table 4.17: Average number of decisions for instances of the BIBD using mzn2smt translation and Yices2 SMT solver

Instances r, v, λ	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
7-3-2	7	7	9	8	**34**	9	6	8
8-4-3	2	2	1	2	**6**	2	2	1
8-4-4	1	1	1	1	1	1	1	1
9-3-1	1	1	1	1	**2**	1	1	1
11-5-2	1	1	1	1	**7**	1	1	1
13-3-1	32	34	44	44	**166**	31	30	37
13-4-1	43	47	57	106	**233**	34	32	64

Table 4.18: Average number of decisions per millisecond for instances of the BIBD using mzn2smt translation and Yices2 SMT solver

4.2.2 BIBD Results using Z3

Z3 did not perform well on most of the instances of the BIBD and *Lex*. Most the results in tables 4.19 and 4.16 are time-outs (>300 seconds). Using Harvey encoding on the directly translated instances, Z3 performed much better than the all the other encodings, though this was not the same on the mzn2smt translated instances. There are no decision tables here because the solver did not provide decisions count for the timed-out results. From the instances 7-3-2, 8-4-3 and 9-3-1 in both tables, constraint reification helped the AND and the OR but had an opposite effect on their CSE variants.

Instances r, v, λ	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey	No *Lex*
7-3-2	77.5	46.6	69.8	21.6	118	104	61.5	**0.44**	2.87
8-4-3	300	221	264	178	230	250	148	**0.76**	5.16
8-4-4	300	300	300	300	300	300	300	300	300
9-3-1	234	155	236	91.3	157	121	38.2	**0.60**	4.76
11-5-2	300	300	300	300	300	300	300	**2.45**	8.05
13-3-1	300	300	300	300	300	300	300	**13.26**	174.02
13-4-1	300	300	300	300	300	300	300	**2.65**	16.25
Arith-Mean	258	232	252	213	243	239	206	**45.7**	73.02
Geo-Mean	238	200	231	161	230	220	161	**3.49**	17.40

Table 4.19: Average solution time (in seconds) for instances of the BIBD using the direct translation and Z3 SMT solver

Instances r, v, λ	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey	No *Lex*
7-3-2	43.71	82.62	45.71	51.40	24.13	1364	13.81	52.73	**2.87**
8-4-3	262	287	174	223	182	164	117	138	**5.16**
8-4-4	300	300	300	300	300	300	300	300	300
9-3-1	175	191	91.3	103	123	72.2	42.3	35.1	**4.76**
11-5-2	300	300	300	300	300	284	299	295	**8.05**
13-3-1	300	300	300	300	300	300	300	300	**174.02**
13-4-1	300	300	300	300	300	300	300	300	**16.25**
Arith-Mean	240	251	216	225	218	222	196	203	**73.02**
Geo-Mean	207	232	179	192	171	199	127	154	**17.40**

Table 4.20: Average solution time (in seconds) for instances of The BIBD using the mzn2smt translation and Z3 SMT solver

4.3 Long Vectors Instances

Our plan was to test the inference capabilities of the eight encodings by running unsatisfiable instances of both the SGP and the BIBD problems, but we could not manage to find suitable such instances. Apart from the instance 8-4-4 of the SGP, all the unsatisfiable instances we tested for both problems were either very easy and solved in a negligible time, or very hard which made them impractical to use in the benchmarks. So we made our own unsatisfiable instances as a simple model that enforces the *Lex* constraint on two vectors, A and B, of a length n, both have the same integer domain of values of 1 to 4. We made the model unsatisfiable by making *Lex* fail at the last two items of both vectors, as shown in (4.2) and (4.3) in the following model.

$$A \leq_{lex} B \tag{4.1}$$

$$1 \leq i \leq n-1 \quad A[i] = 4 \tag{4.2}$$

$$B[n] = A[n] - 1 \tag{4.3}$$

4.3.1 Long Vectors Results

Results for the Long Vectors tests are arranged in four tables (4.21, 4.22, 4.23 and 4.24) representing results for two SMT solvers, Yices2 and Z3, using the mzn2smt and the direct translations. Looking at the results in general, it is clear that the increasing formula size negatively affected the performance in the cases of AND, OR and ANDOR, and caused the Out-of-Memory problem. Common sub-expression elimination greatly helped in reducing formula size thus eliminated the Out-of-Memory problem, as it is clear in tables 4.25 and 4.26. Apart from AND CSE in Yices2 results, constraint reification had unfavourable effect on all encodings, and magnified the Out-of-Memory problem in AND, OR and ANDOR because of the additional variables the constraint reification introduces. The source of Out-of-Memory problem was the mzn2fzn tool that we used in our mzn2smt translation pipeline.

Vector Size	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey	ANDOR
500	0.29	0.13	0.25	0.09	**0.01**	0.02	**0.01**	0.02	0.49
1000	1.30	0.60	1.02	0.35	**0.02**	0.03	**0.02**	0.04	2.07
2000	5.99	3.27	4.11	1.40	0.05	0.07	**0.04**	0.08	8.40
2500	9.98	5.69	6.50	2.16	**0.05**	0.08	**0.05**	0.09	15.91
3000	15.40	9.25	9.25	3.08	**0.07**	0.10	**0.07**	0.12	20.80
Average	6.59	3.79	4.23	1.42	**0.04**	0.06	**0.04**	0.07	9.54

Table 4.21: Average solution time (in seconds) for unsat instances of *Lex* between two vectors using the direct translation and Yices2 SMT solver

Vector Size	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey	ANDOR
500	0.17	0.09	0.26	0.12	0.11	0.10	0.08	**0.02**	0.33
1000	OOM	0.32	OOM	0.48	0.42	0.36	0.30	**0.04**	OOM
2000	OOM	1.26	OOM	2.03	1.73	1.71	1.19	**0.07**	OOM
2500	OOM	1.99	OOM	3.44	2.94	3.07	2.18	**0.09**	OOM
3000	OOM	3.00	OOM	5.89	4.73	5.09	3.09	**0.11**	OOM
Average		1.33		2.39	1.99	2.06	1.37	**0.07**	

Table 4.22: Average solution time (in seconds) for unsat instances of *Lex* between two vectors using mzn2smt translation and Yices2 SMT solver

Vector Size	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey	ANDOR
500	0.53	0.14	0.34	**0.05**	0.10	0.33	**0.05**	0.11	0.63
1000	2.03	0.42	1.35	**0.09**	0.16	1.26	0.10	0.21	2.62
2000	7.68	1.40	4.73	**0.14**	0.21	4.19	0.16	0.38	9.27
2500	12.46	2.41	8.70	**0.20**	0.28	7.95	0.23	0.54	18.18
3000	31.39	3.57	13.34	**0.24**	0.33	11.87	0.27	0.64	27.86
Average	10.82	1.59	5.69	**0.14**	0.21	5.12	0.16	0.38	11.71

Table 4.23: Average solution time (in seconds) for unsat instances of *Lex* between two vectors using the direct translation and Z3 SMT solver

Vector Size	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey	ANDOR
500	0.35	0.17	0.21	0.08	0.27	0.39	**0.07**	0.12	0.42
1000	OOM	0.58	OOM	0.15	0.99	1.44	**0.13**	0.23	OOM
2000	OOM	2.67	OOM	0.34	4.10	5.98	**0.26**	0.50	OOM
2500	OOM	4.56	OOM	0.49	6.57	10.13	**0.32**	0.60	OOM
3000	OOM	17.39	OOM	0.63	9.55	14.21	**0.39**	0.74	OOM
Average		5.07		0.34	4.30	6.43	**0.24**	0.44	

Table 4.24: Average solution time (in seconds) for unsat instances of *Lex* between two vectors using mzn2smt translation and Z3 SMT solver

4.3.2 Instances and Encoding Size

From the Long Vectors results, it is clear that formula size played a major rule in affecting the performance of both solvers on some instances, this becomes more apparent with each increment in instances size. The smallest and with the best results is the formula that produced by the directly translated ROR encoding. Though this does not affect all encodings by the same factor. It seems that the AND encoding is more sensitive to formula size than the others, for instance a directly translated OR encoding performs twice as better as a formula of the same size of a directly translated AND. This is interesting because the AND encoding is closer to the natural SAT's CNF than the OR which is basically a Disjunctive Normal Form (DNF). Both the AND and the OR have a quadratic formula growth.

Vector Size	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey	ANDOR
500	125	2	125.75	2	1	4	2	1	251
1000	500	5	501.50	5	2	8	4	2	1002
2000	2001	10	2003.00	10	4	17	8	4	4004
2500	3126	12	3128.75	12	5	22	10	5	6255
3000	4501	15	4504.50	15	6	26	12	6	9006
Average	2051	9	2053	9	4	16	7	4	4104

Table 4.25: Number of occurrences of atoms (divided by 1000) for the instances of *Lex* between two vectors using the direct translation

Vector Size	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey	ANDOR
500	128	4	257	6	6	9	5	4	386
1000	506	8	1015	13	12	18	10	8	1521
2000	2012	17	4030	26	24	36	20	16	6043
2500	3141	22	6287	33	30	45	25	20	9429
3000	4519	26	9045	39	36	54	30	24	13564
average	2062	16	4127	23	22	32	18	14	6189

Table 4.26: Number of occurrences of atoms (divided by 1000) for the instances of *Lex* between two vectors using mzn2smt translation

4.4 BIBD All Solutions Benchmarks

After testing the encodings on large instances of two vectors, we decided to run further tests on hard instances of a problem. We choose to do this by trying to find all solutions for instances of the BIBD. This done by modifying our benchmarking script to run the Yices2 to solve an instance in a loop, and after each run that results in a solution the script feeds back the inverted solution to the instance to rule out that solution, this continues till the solver gives unsat, which means that all solutions has been found.

Tables 4.27 and 4.30 show the timing for those instances where Yices2 was able to find all solutions without timing-out. On the instance 13-3-1 Yices2 was unable to find all solutions before the time-out (300 seconds), so instead of the timings, we reported number of solutions found using each encoding in tables 4.28 and 4.31. As expected, without symmetry breaking finding all solutions would take longer time in most cases. Here, table 4.29 shows number of solutions found by Yices2 in around 300 seconds.

The results reflect close performances between AND, AND CSE, Alpha and AlphaM, with a marginal lead for AND in some cases. Apart form the cases of OR and ROR, the mzn2smt translation didn't improve performance compared to the direct translation. Table 4.29 reflects the variation of difficulty levels between the instances we used.

Instances r, v, λ	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
7-3-2	6.32	**6.24**	10.07	9.49	10.21	**6.24**	7.91	6.53
8-4-3	**103.24**	113.87	153.62	149.29	157.15	105.22	105.24	155.31
9-3-1	2.68	**2.15**	3.78	3.98	3.41	2.50	2.31	2.58
11-5-2	1.29	1.27	10.40	4.59	1.48	**1.04**	1.08	1.91
13-4-1	**26.21**	34.38	88.41	53.47	52.05	34.86	30.18	52.60
Arith-Mean	**27.95**	31.58	53.26	44.16	44.86	29.97	29.34	43.79
Geo-Mean	**9.00**	9.22	22.19	16.90	13.33	9.02	9.11	12.13

Table 4.27: Average solution time (in seconds) for finding all solutions for instances of the BIBD using the direct translation and Yices2 SMT solver

Instances r, v, λ	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
Solutions in 300s for 13-3-1	14	13	12	13	15	17	15	12

Table 4.28: Average number of solutions found in 300 seconds for the instance 13-3-1 of the BIBD using the direct translation and Yices2 SMT solver

Encoding	7-3-2	8-4-3	9-3-1	11-5-2	13-3-1	13-4-1
No *Lex*	641	309	426	148	14	62

Table 4.29: Number of solutions found in 300 seconds for instances of the BIBD without *Lex* constraint and using Yices2 SMT solver

Instances r, v, λ	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
7-3-2	**8.07**	9.27	12.57	11.71	12.23	10.87	11.70	8.54
8-4-3	153.13	147.29	**146.95**	181.68	152.31	169.34	200.76	168.04
9-3-1	3.82	2.99	4.32	4.36	**2.60**	3.79	4.25	3.21
11-5-2	**1.12**	1.38	3.72	3.73	1.28	1.33	1.51	7.52
13-4-1	**28.67**	72.94	58.85	86.18	38.20	56.74	31.33	81.09
Arith-Mean	**38.96**	46.77	45.28	57.53	41.32	48.41	49.91	53.68
Geo-Mean	**10.87**	13.27	17.72	19.72	11.88	13.94	13.65	19.49

Table 4.30: Average solution time (in seconds) for finding all solutions for instances of the BIBD using the mzn2smt translation and Yices2 SMT solver

Instances r, v, λ	AND	AND CSE	OR	OR CSE	ROR	Alpha	AlphaM	Harvey
Solutions in 300s for 13-3-1	14	14	15	14	15	14	14	13

Table 4.31: Average number of solutions found in 300 seconds for the instances 13-3-1 of the BIBD using the mzn2smt translation and Yices2 SMT solver

Chapter 5

Evaluation and Conclusions

Contrary to expectations, the results show that when searching for a single solution, not using any of the eight *Lex* encodings (No Lex) is better on most of the instances of the SGP and the BIBD we used. But still, using *Lex* would perform better in solving unsatisfiable instances or when all possible solutions were required as we saw in all solutions tests. In many cases, it is difficult to know whether an instance of a problem is satisfiable or not, this makes it difficult to have prior knowledge of the instances which would perform better without symmetry breaking using *Lex*. For that, we decided to evaluate the encodings by the differences their performances ignoring the performance of the No Lex.

The results, in general, show that the Recursive OR encoding has the best results on most of the tables. However, all the encodings preformed differently on different instances of the same problem, and even sometimes on different samples of the same instance. For example, on the SGP, the Recursive OR encoding did very well on 7 out of the 10 instances, but it was 100 times worse than its nearest competitor on one of those instance of the problem, it is also did not perform as well in all solutions test. This behaviour promotes the idea of using special algorithms for *Lex* [6] instead of the decomposition to tackle symmetry. Also, apart from the solving time, both SMT solvers did not produces statistics for most of the unsat long vectors instances we used, that is why there are no tables for number of decisions for the long vectors instances. The improvement in performance in the case of Harvey encoding when using the mzn2smt translation and Yices, supports Bofill's observation [12] that more logical component over theory in a formula helps to improve solving time, though this true only in case of Yices2 but not with Z3 as it is clear from tables 4.19 and 4.20, this implies that this behaviour is dependant on the solving algorithm. The Out-of-Memory problem in the results demonstrated the limitations that the AND, OR and the ANDOR encodings have when dealing with relativity sizeable problems. Finally, in the majority of cases, the performance of the directly translated instances was better than those ones which went through Bonfill's fzn2smt translator.

5.1 Future Work

Without closely examining how SMT solvers approach the solving process, it is hard to get enough explanations from the results alone. A future line of research could be to study how SMT solvers internally handle each encoding, this can be done by examining the code of some the open source SMT solvers and by building a framework that monitors a solver during runtime. Alongside this, doing more benchmarks using different CSP problems and SMT solvers could help to find clearer patterns from the results.

Abbreviations

BCP Boolean Constraints Propagation

BIBD Balanced Incomplete Block Design

CDCL Conict Driven Clause Learning

CNF Conjunctive Normal Form

CSE Common Sub-expression Elimination

CSP Constraint Satisfaction Problems

DNF Disjunctive Normal Form

DPLL Davis, Putnam, Logemann and Loveland

GAC Generalised Arc Consistency

ROR Recursive OR

SB Symmetry Braking

SGP Social Golfer Problem

References

[1] MiniZinc and FlatZinc. `http://www.minizinc.org/`, 2014.

[2] SRI International. `http://www.sri.com/`, 2015.

[3] Fadi A Aloul, Karem A Sakallah, and Igor L Markov. Efficient symmetry breaking for boolean satisfiability. *Computers, IEEE Transactions on*, 55(5):549–558, 2006.

[4] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In *Automated DeductionCADE-18*, pages 195–210. Springer, 2002.

[5] Roderick D Ball. Incomplete block designs for the minimisation of order and carry-over effects in sensory analysis. *Food Quality and Preference*, 8(2):111–118, 1997.

[6] Milan Bankovic and Filip Maric. An alldifferent constraint solver in smt. In *8th International Workshop on Satisfiability Modulo Theories*, 2010.

[7] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[8] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[9] Clark Barrett and Cesare Tinelli. CVC4. `http://cvc4.cs.nyu.edu/web/`, 2015.

[10] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. *Handbook of satisfiability*, 185:825–885, 2009.

[11] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.

[12] Miquel Bofill, Josep Suy, and Mateu Villaret. A system for solving constraint satisfaction problems with SMT. In *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 300–305. Springer, 2010.

[13] David Cok, David Deharbe, and Tjark Weber. SMT-COMP 2014. `http://www.smtcomp.org`, 2014.

[14] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[15] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. *KR*, 96:148–159, 1996.

[16] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[18] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[19] Leonardo De Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Automated DeductionCADE-18*, pages 438–455. Springer, 2002.

[20] David Detlefs, Greg Nelson, and James B Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[21] Bruno Dutertre and Leonardo De Moura. A fast linear-arithmetic solver for DPLL (T). In *Computer Aided Verification*, pages 81–94. Springer, 2006.

[22] Bruno Dutertre and Leonardo De Moura. The Yices SMT solver. *Tool paper at http://yices.csl.sri.com/tool-paper.pdf*, 2(2), 2006.

[23] Pierre Flener, Alan M Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix mod-

els. In *Principles and Practice of Constraint Programming-CP 2002*, pages 462–477. Springer, 2002.

[24] Pierre Flener, Justin Pearson, Meinolf Sellmann, and Pascal Van Hentenryck. Static and dynamic structural symmetry breaking. In *Principles and Practice of Constraint Programming-CP 2006*, pages 695–699. Springer, 2006.

[25] Eugene C Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *AAAI*, volume 91, pages 227–233, 1991.

[26] Alan M Frisch and Warwick Harvey. Constraints for breaking all row and column symmetries in a three-by-two matrix. In *Proc. Symcon*, 2003.

[27] Alan M Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Propagation algorithms for lexicographic ordering constraints. *Artificial Intelligence*, 170(10):803–834, 2006.

[28] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Theory and Applications of Satisfiability Testing-SAT 2006*, pages 252–265. Springer, 2006.

[29] Ian P Gent. Arc consistency in SAT. In *ECAI*, volume 2, pages 121–125, 2002.

[30] Ian P Gent, Patrick Prosser, and Barbara M Smith. A 0/1 encoding of the gaclex constraint for pairs of vectors. In *Notes of the ECAI-02 Workshop W9 Modelling and Solving Problems with Constraints*, 2002.

[31] Ian P Gent and Barbara Smith. *Symmetry breaking during search in constraint programming*. Citeseer, 1999.

[32] Enrico Giunchiglia, Roberto Sebastiani, Fausto Giunchiglia, and Armando Tacchella. SAT vs. Translation based decision procedures for modal logics: a comparative evaluation. *Journal of Applied Non-Classical Logics*, 10(2):145–172, 2000.

[33] Fausto Giunchiglia and Roberto Sebastiani. *Building decision procedures for modal logics from propositional decision proceduresthe case study of modal K*. Springer, 1996.

[34] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a Parallel SAT Solver. *JSAT*, 6(4):245–262, 2009.

[35] Warwick Harvey. Warwick's Results Page for the Social Golfer Problem. `http://www.icparc.ic.ac.uk/ wh/golf/`, 2014.

[36] Ian Horrocks. The fact system. In *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 307–312. Springer, 1998.

[37] Warren A Hunt and Steven D Johnson. *Formal methods in computer-aided design.* Springer, 2000.

[38] Chris Jefferson, Bilal Hussain, and Chris Jefferson. CSPLib: A problem library for constraints. `http://www.csplib.org/`, 2015.

[39] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering*, 11(4):403–434, 2004.

[40] Zeynep Kiziltan. Symmetry breaking ordering constraints. *AI Communications*, 17(3):167–169, 2004.

[41] Hakan Kjellerstrand. hakank's Home Page. `http://www.hakank.org/`, 2014.

[42] Sharad Malik and Lintao Zhang. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009.

[43] Panagiotis Manolios, Marc Galceran Oms, and Sergi Oliva Valls. Checking pedigree consistency with PCS. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 339–342. Springer, 2007.

[44] Joao Marques-Silva. Practical applications of Boolean satisfiability. In *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pages 74–80. IEEE, 2008.

[45] João P Marques-Silva and Karem A Sakallah. GRASP: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.

[46] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[47] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[48] SD Prestwich. Balanced incomplete block design as satisfiability. In *Proceedings of the 12th Irish Conference on Artificial Intelligence and Cognitive Science*, 2001.

[49] Jean-Francois Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Methodologies for Intelligent Systems*, pages 350–361. Springer, 1993.

[50] Karem A Sakallah. Symmetry and Satisfiability. *Handbook of Satisfiability*, 185:289–338, 2009.

[51] Roberto Sebastiani. Lazy satisfiability modulo theories. 2007.

[52] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design*, pages 127–144. Springer, 2000.

[53] Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Electronic Notes in Discrete Mathematics*, 9:19–35, 2001.

[54] Robert E Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM (JACM)*, 26(2):351–360, 1979.

[55] Alexander Smith, Andreas Veneris, M Fahim Ali, and Anastasios Viglas. Fault diagnosis and logic debugging using Boolean satisfiability. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(10):1606–1621, 2005.

[56] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.

[57] Ian N Wakeling and Dominic Buck. Balanced incomplete block designs useful for consumer experimentation. *Food quality and preference*, 12(4):265–268, 2001.

[58] Toby Walsh. Sat v csp. In *Principles and Practice of Constraint Programming–CP 2000*, pages 441–456. Springer, 2000.

[59] Toby Walsh. General symmetry breaking constraints. In *Principles and Practice of Constraint Programming-CP 2006*, pages 650–664. Springer, 2006.

[60] Hantao Zhang. Generating college conference basketball schedules by a SAT solver. In *Proceedings of the fifth international symposium on the theory and applications of satisfiability testing*, pages 281–291, 2002.

[61] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4):543–560, 1996.

[62] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.

Index