Astra: Evaluating Translations from Alloy to SMT-LIB

by

Ali Abbassi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

We present a variety of translation options for converting Alloy to SMT-LIB via Alloy's Kodkod interface. Our translations, which are implemented in a library that we call Astra, are based on converting the set and relational operations of Alloy into their equivalent in typed first-order logic (TFOL). We investigate and compare the performance of an SMT solver for many translation options. We have three translation axes, and in total, twelve different combinations. We compare using only one universal type to recovering Alloy type information from the Kodkod representation and using multiple types in TFOL. We compare a direct translation of the relations to predicates in TFOL to one where we recover functions from their relational form in Kodkod and represent these as functions in TFOL. We compare representations in TFOL with unbounded scopes to ones with bounded scopes, either pre or post quantifier expansion.

We propose characteristics for classifying problems, which we hypothesize affect the performance. We provide a set of test cases with different characteristics, and by testing our translation on our tests, we create a statistical model to correlate characteristics to the performance of different translation options. We propose hypotheses regarding SMT solvers and modelling guidelines, and test them based on our empirical results. Our results across all these dimensions provide directions for portfolio solvers, modelling improvements, and optimizing SMT solvers.

At the end, we present a set of questions that suggest future work. These questions are based on results we could not justify or find a reason for. The subjects of these questions are SMT solvers and modelling optimizations.

Acknowledgements

I would first like to thank my supervisor, Dr. Nancy A. Day, because without her guidance, this thesis would have not been possible. She gave me opportunities for which I will always be thankful. She taught me countless lessons, which I will always remember. Most importantly, I want to thank her for trusting and supporting me and giving me the freedom to work on what I enjoy. I feel honoured to work with her.

I would also like to thank the Dr. Derek Rayside, who gave us constructive suggestions and helped us throughout this work. I would also like to thank Dr. Richard Trefler for reviewing my thesis. I want to thank Dr. Amirhossein Vakili, for his support and guidance throughout my time in Waterloo.

I would like to thank my colleagues, Jose Serna, and Amin Bandali, with whom I had the pleasure of working. I want to thank all other people who helped me get through my master's studies successfully.

Finally, I would like to thank my family, whose support has always been with me; I have always felt their presence in my life regardless of the physical distance between us. I am very grateful for having them with me.

Table of Contents

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Contributions . . . 3

1.1.1 Translation from relational FOL to TFOL . . . 4

1.1.2 Testing a set of benchmarks for performance results . . . 4

1.1.3 Test results and modelling guidelines . . . 5

1.2 Thesis overview . . . 6

2 Background 7

2.1 Finite Model Finding . . . 7

2.2 Alloy . . . 8

2.2.1 Kodkod . . . 10

2.3 Fortress . . . 13

2.4 Z3 . . . 15

2.5 Summary . . . 17

3 Astra 18

3.1 Overview . . . 18

3.2 Step 1: Declaring SMT types . . . 20
3.3 Step 2: Declaring SMT functions and relations . . . 21
3.4 Step 3: Formula translation . . . 23
3.5 Step 4: Scope translation . . . 28
3.5.1 Unscoped . . . 29
3.6 Step 5: SMT solver . . . 29
3.7 Implementation . . . 29
3.8 Related work . . . 30
3.9 Summary . . . 31

4 Performance Testing **32**
4.1 Astra options . . . 32
4.2 Characteristics of the models . . . 33
4.3 Case studies . . . 35
4.4 Performance results . . . 37
4.4.1 Statistical model . . . 37
4.5 Evaluation . . . 39
4.5.1 Hypotheses . . . 39
4.5.2 Guidelines for suitable analysis combination . . . 40
4.5.3 Discussion . . . 42
4.6 Summary . . . 42

5 Related Work **43**

6 Conclusion **45**

References **46**

List of Tables

2.1 Test results for comparison between Kodkod and Fortress 16

3.1 Types, functions and relations after detection in pass 1 for typed option. . 21
3.2 Types, functions and relations after detection in pass 1 for untyped option. 21
3.3 Function and predicate declarations after the pass 1 for typed option . . . 22
3.4 Function and predicate declarations after the pass 1 for untyped option . . 24

4.1 Translation combinations 33
4.2 Characteristics of the tests 35

List of Figures

1.1 Different translation dimensions and options. 5

2.1 A small Alloy example 9
2.2 Partial Kodkod input syntax taken from Torlak *et al.* [22] 11
2.3 Parts of Kodkod translation pseudo code of the Alloy example 12
2.4 Fortress input syntax taken from Vakili *et al.* [24] 14

3.1 Steps in translation. 19
3.2 Formula translation for typed option. 25
3.3 Formula translation for untyped option. 27

4.1 Characteristics of tests. 36
4.2 Performance results for tests. 38
4.3 Decision tree 41

Chapter 1

Introduction

In the software development process, especially for critical systems, it is vitally important to find errors and conflicts as early as possible in the process to reduce the cost of development and maintenance, and eliminate the risk of failure that may cause extensive human and financial damage. Formal methods offer a variety of ways by which these systems can be modelled and checked formally, in an abstract and concise way.

First-order logic (FOL) is one of the ways to model the system and inspect its correctness. FOL is largely used in modelling because of the abstraction level and conciseness it provides. However, the decision procedure for FOL satisfiability is undecidable, meaning that automatic tools cannot guarantee the termination of the decision procedure. Also, as the complexity of systems grows, it becomes harder to prove properties for them with theorem provers, which need manual guidance. Thus, it is necessary to come up with ways to check models in FOL, at least partially, to gain more confidence in the model.

Finite model finding is one of the suggested solutions to deal with the undecidability problem following Jackson's small scope hypothesis [18], which states that a large proportion of errors can be found by testing the model within a relatively small scope. There are several tools over different languages that are focused on limited domains rather than checking for unbounded scope, so that the analysis is guaranteed to return an answer, given adequate computing resources. A popular finite model finding tool is the Alloy Analyzer [17]. While the answer for the limited scope problem might not be the answer to the main problem, it can increase confidence in the proposed model.

Alloy [17] is a language developed by Jackson. This language is based on relational FOL and is powerful and effective for modelling and describing systems formally. Relational FOL

is FOL with relational and set theory operations. The Alloy Analyzer, translates the model into Kodkod, which is the intermediate language between core Alloy and a SAT solver.

Kodkod is a finite model finder for relational FOL based on SAT solvers. Kodkod does not have a type system, instead, it uses relations to specify different aspects of the model. Types are translated to unary relations, and different types of functions and relations are considered as n-ary relations, with extra constraints if needed. Translating functions as constrained relations can cause inefficiency in the scoped analysis particularly since the size of relations can grow much faster than functions when the size of the scope grows.

Fortress is a finite model finder for Typed First-Order Logic (TFOL). Fortress uses SMT solvers [5] instead of SAT solvers, which means it supports types and can use the help of theories to its advantage in comparison with Kodkod. Additionally, Fortress is expected to have better results than Kodkod since, as shown in Vakili *et al.* [24], Fortress had better analysis performance than Kodkod on Alloy models when tested on the same benchmark. Consequently, using Fortress for Alloy language problems may increase its efficiency of the analysis time.

We can inspect unscoped models using an SMT solver directly. In these cases, due to the presence of quantifiers in the models, the decision procedure is not guaranteed to terminate. Having this option besides finite model finding in Alloy, gives the user the ability to check whether the problems can be solved for all scopes or not.

Another way to use an SMT solver to solve Alloy language problems is to ask an SMT solver to search only with a limited scope. We can do this by adding range formulas where we put constraints on the number of atoms for each type, meaning that each variable of each type can only be a member of a finite set of constants. This way, the problem is syntactically undecidable, but it has only finite scope solutions.

SMT solvers use the SMT-LIB [4] language as the input language, which has a relatively simple structure. The SMT-LIB language is based on TFOL.

El Ghazi *et al.* [14], describe a translation directly from Alloy to the SMT solver Yices [11]. One of the advantages that Yices has compared to the SMT-LIB language is support for subtypes, which are extensively used in Alloy. This work is for unscoped problems.

El Ghazi *et al.* [12] proposed a direct translation from Alloy to TFOL. Another translation from relational FOL to TFOL is the translation presented in Ulbrich *et al.* [23]. The former is a translation from Alloy to SMT-LIB only for unscoped problems, which does not guarantee the decision procedure termination. The latter, describes a translation from Alloy to the KeY theorem prover [6] for first-order logic to check Alloy models over un-

bounded scopes. However, theorem provers do not always automatically solve a problem, and often require manual guidance of an expert.

Bansal *et al.*[2] introduces a new theory for solving relational FOL (including set cardinality) of finite scope problems in SMT solvers. This theory is a calculus for relational logic for finite sets in SMT. Since the scope of problems is finite, the decision procedure is guaranteed to terminate in this work.

Reynolds *et al.* [20] propose a theory called Finite Cardinality Constraints (FCC) for doing finite model finding within an SMT solver. But for this theory to fully support the Alloy language, relational FOL theories are needed. This work is an approach for finite model finding, in which the satisfiability check is decidable.

Finally, Alloy2B[19] is a tool that translates Alloy models to the B language [1], making a variety for B tools available for use on Alloy models including model checkers and interactive proof tools for examining a model of unbounded scope.

Compared to the above work, in this thesis, we investigate the performance of a number of translation options for converting Alloy to typed FOL (TFOL). We call our library Astra (Alloy to SMT-LIB translation). To enable future integration with the Alloy Analyzer, we start from the Kodkod interface. We compare using only one universal type to recovering Alloy type information from the Kodkod representation and using multiple types in TFOL. We compare a direct translation of the relations to predicates in TFOL to one where we recover functions from their relational form in Kodkod and represent these as functions in TFOL. We compare representations in TFOL with unbounded scopes to ones with bounded scopes, either pre or post quantifier expansion.

1.1 Contributions

This thesis is focused on improving and analyzing the performance of the verification of Alloy models by translating the formulas from Kodkod to TFOL. We describe different options for translation, which make it possible to experiment with different factors that may make a difference in analysis performance of Alloy models. We propose hypotheses about our translation combinations and SMT solvers and test them.

> **Thesis statement:** Alloy can be linked to SMT solvers via its Kodkod interface by translating Kodkod formulas to typed first-order logic. There are a variety of equivalent ways to do this translation. There is empirical and

explanatory evidence that model characteristics can be correlated with translation combinations to achieve the best performance amongst the translation combinations.

We run our performance tests using the SMT solver Z3 [10] to compare the results of all of our translation options to each other and Kodkod. We categorized our tests by characteristics of the model such as depth of quantifiers, number of types, the use of join, number of functions, *etc.* and discuss whether there are any meaningful correlations between model characteristics and our translation options.

Our results are useful to anyone who uses Alloy for verification. We have not yet implemented transitive closure, cardinality of sets, or support for built-in Alloy types. Our contribution comparing all of these translation options provides directions for portfolio solvers, modelling improvements, and optimizing SMT solvers.

We next explain the contributions of this thesis.

1.1.1 Translation from relational FOL to TFOL

Relational FOL and TFOL are equivalent in terms of expressiveness. We propose different equivalent translations, based on factors that we hypothesize can impact analysis performance. Our translations are close to what has been proposed in the work by Ulbrich *et al.* [23] and El Ghazi *et al.* [12], but they differ somewhat, which we describe in Chapter 3.

As presented in Figure 1.1 our different translations have three axes: the type axis, the function axis, and the scope axis. The options on the type axis are typed and untyped. There are two different choices on the function axis, functions and predicates. Lastly, we have Fortress, SMT finite model finding (SMT FMF), and unscoped for the scope axis. Since the translation axes are independent, the resulting total possible combinations of the options is equal to twelve. Because the translation starts from Kodkod, we had to reverse engineer the derived structures and formulas to find the original properties of the model. Our translation is easier than if we wanted to translate directly from Alloy due to the simpler structure of the Kodkod language compared to Alloy, but is made harder because of the reverse engineering required.

1.1.2 Testing a set of benchmarks for performance results

We have eleven Alloy models that are tested for different scopes for a total of twenty-nine tests. Most tests are not satisfiable because these tend to be harder problems for solvers,

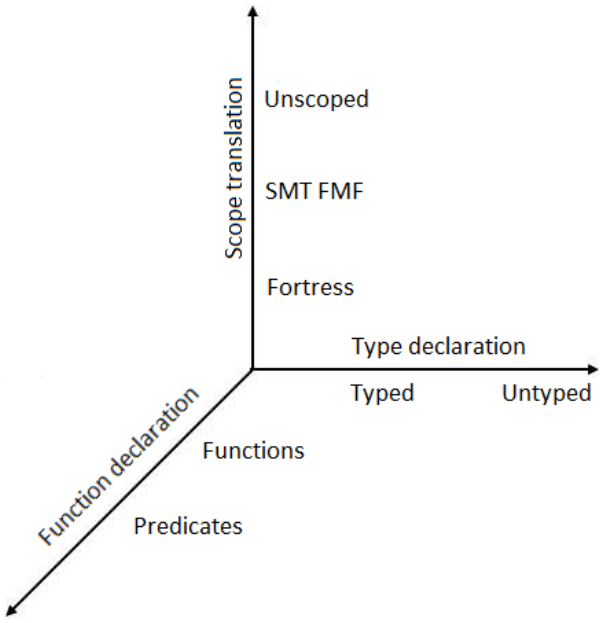

Figure 1.1: Different translation dimensions and options.

but a few tests have satisfying solutions. These models originated from Kodkod benchmarks [22], which were also used to test Fortress, and we created some additional models to ensure that we have models that contained interesting instances of several characteristics. We introduce a set of characteristics that we hypothesize can impact the performance analysis, and measured them on our models. We test models that represent different combinations of these characteristics, and we draw conclusions based on our tests that can be applied for finding the best option for each model.

1.1.3 Test results and modelling guidelines

We test Astra with our test cases, and present the results for the interesting translation combinations. Interesting combinations are the ones that perform the best on at least one test. Out of the total twelve combinations, we found four interesting ones that are analyzed and discussed.

We then evaluate our hypotheses, and explain them using our empirical data. These hypotheses provide modellers with guidelines that can improve the performance of the verification of Alloy models. These guidelines can also be used to understand and improve SMT solvers. In addition to inspecting out hypotheses, we try to find statistical models

based on performance times of different options and model characteristics. These statistical models indicate which characteristics affect performance time the most. Based on our statistical models, a user can find the option that is most likely to solve the problem faster. We believe this process can be automated so that the tool can find the best option based on the characteristics of the model without user's input.

At the end, we present several questions, based on Astra's behavior in different combinations. These are the questions that we could not find the answer to in this thesis, and these questions suggest future work regarding improving SMT solvers' performance and creating guidelines for modelling.

1.2 Thesis overview

This thesis is organized as follows: In the next chapter, a brief background on finite model finding, Alloy, Kodkod, Fortress, and Z3 is provided. Our translation is explained in Chapter 3. Chapter 4, discusses the tests and their characteristics. In Chapter 5, we present our test results and conclusions drawn from them. Related work is presented in Chapter 6. Lastly, we concluded the thesis in Chapter 7.

Chapter 2

Background

In this section, we present a brief overview of finite model finding, Alloy, Kodkod, Fortress, and Z3.

2.1 Finite Model Finding

First-order logic satisfiability checking is inherently undecidable, meaning that there is no algorithm that can solve every problem in FOL. Therefore, there cannot exist an automatic tool for solving all problems in FOL. The complexity of the models grow as the problems grow in complexity, and additionally, manual theorem proving requires expertise and is difficult to use in industrial settings. Automatic tools, such as SMT solvers, often come up with no answer to our question when they face a complicated problem. To avoid this problem, finite model finding approaches can be used.

In finite model finding, by limiting the scope of the universe, only the finite solutions are considered. The decision procedure for finding finite solutions is decidable. Of course, solving the latter does not mean that the original problem has been solved, but it is helpful in cases where there is no answer to the original problem. If a solution is found, it is a solution to the original problem, but if the result is UNSAT, then it is only correct for the limited scope. Relying on Jackson's small scope hypothesis [18], more confidence in the model can be gained through this process.

Finite model finding is used in different settings, such as proving theorems in finite algebra, checking lightweight formal specification, finding counterexample to tentative theorems in interactive proof assistance, bounded verification of code and memory models,

and declarative configuration and execution. Many different tools have been developed in this area, such as Forge [15], Fortress [24], Kodkod [22].

2.2 Alloy

Alloy is a declarative language for modelling complex systems. Its core logic is typed relational FOL. Alloy is an effective lightweight language, and its simplicity, abstraction, modularity, good tool support, and large community make it a suitable candidate for being used in different contexts. The Alloy Analyzer, which is the tool for the Alloy language, is open source, and the users and developers can access and contribute to the tool.

An Alloy model consists of three main parts, declarations, constraints, and assertions. Types are declared as signatures. Constraints are parts of an Alloy model that limits the model, and are expressed by facts and predicates. Lastly, assertions are stated by the `check()` command on certain predicates. Alloy also allows users to use the `run()` command for checking the consistency of the model, with a predicate, or alone.

Alloy has good syntax support for types and inheritance, which increases the modularity and conciseness of the model. Through the `sig` and `extends` keywords, complex type hierarchies can be defined in Alloy. Relations and functions are defined within `sig` blocks. Other constraints and structures can be defined as `pred` for predicates, `fact` for invariant constraints, and `fun` functions. Facts are assumed to be always true about the model. Predicates are true wherever they are used. Functions have return values and can make a model more modular.

Figure 2.1 presents a small Alloy example. Lines 1 to 13 are the declarations. The type `A` is an abstract type, meaning that it is equal to the union of all its subtypes, and it does not contain any member that is not a member of one of its subtypes. Lines 14 to 27 describe the invariants of the model. Lastly, line 28, sets the scope of the problem and runs the Alloy Analyzer to find a satisfying model.

The declarations include the declaration of types and subtypes, relational functions, and relations. In this work, for conciseness, we use function instead of relational function. In the example, types `B` and `C` are declared to be subtypes of abstract type `A`. Function `id` is defined within the `A` signature, in line 3. It is a function, because of the multiplicity assigned to it, `one`. We can also find a relation in line 11, since the multiplicity of the relation is `set`. Other multiplicities are `lone` and `some`, which mean partial function and relation respectively. The difference between `set` and `some` is that each member of domain

```
1  abstract sig A {
2      // Function, id: A -> ID
3      id: one ID
4  }
5  sig B extends A {
6      // Function, toC: B -> C
7      toC: one C
8  }
9  sig C extends A {
10     // Relation, toB: C <-> B
11     toB: set B
12 }
13 sig ID {}
14 // Each atom in A has a unique id. (one-to-one function)
15 fact uniqueID {
16     all a, a': A | a.id = a'.id => a = a'
17 }
18 // Each two different atoms in B have pointers to different Cs
19 // (Different way to specify that toC is a one-to-one function)
20 fact uniqueB {
21   all b, b': B | not (b = b') => not (b.toC = b'.toC)
22 }
23 // Relational images of different atoms of type B in toB,
24 // are disjoint.
25 fact uniqueC {
26     all c, c': C | not (c = c') =>
27         not (some b: B | b in (c.toB & c'.toB))
28 }
29 run {} for exactly 6 A, exactly 3 B, exactly 3 C, exactly 6 ID
```

Figure 2.1: A small Alloy example

must be mapped to at least one member of the range when `some` is chosen, however, this is not true for `set`.

Invariants are specified within `fact` blocks. Quantifiers can be used to constrain the model. The universal quantifier is `all`, used in lines 16, 19, and 25, and the existential quantifier is `some`, used in line 26. The join operation, `.`, is extensively used in Alloy, to represent the relational join operation, and function/relation application. For example, in line 16, since `id` is a function, `a.id` behaves like a function application and means $id(a)$. Other relational operations in Alloy include union (`+`), intersection (`&`), set difference (`-`), relational transpose (`~`), and transitive closure (`^`).

Finally, to specify the scope of the model, we can either use `exactly`, which means the problem will be only solved for models with the exact scope as specified, or we can remove it and have our model checked against at most the numbers specified for each type. We only deal with exact scopes in Astra.

2.2.1 Kodkod

Kodkod [22] is the intermediate tool that lies between Alloy and SAT solvers. Unlike Alloy, Kodkod does not have constructs for types or inheritance. The type and function declarations are represented as relations in Kodkod. Types are declared as unary relations in Kodkod, and functions and relations are declared as n-ary relations with extra constraints. The rest of the operations are similar to Alloy's. The subset of Kodkod's syntax that is supported in this work is shown in Figure 2.2. We omit comprehension, if-then-else, and transitive closure.

Kodkod has a Java interface, and it does not have an actual input language. However, there is an option in the Alloy Analyzer to generate Java code representing the Alloy model that interfaces with Kodkod. This Java code consists of three data structures that carry all the information about the Alloy model: a formula; all the generated atoms called the "atomlist"; and the generated universe for all relations called the "bounds". Using these data structures, we can derive the original model.

Kodkod's pseudo code for the example in Figure 2.1 is presented in Figure 2.3. The keyword "this" is because of the way Kodkod translates types and chooses names for variables to differentiate them when the same name is chosen for different constructs. For example, if a variable is passed as an argument to a predicate with the name `P`, the variable would have "P" appended to them. The keyword "univ" is the set of all atoms. Also, each function and relation that is declared in a signature block has the name of the signature appended to its beginning. For example, function "id" in Alloy becomes "A.id" in Kodkod.

$$
\begin{aligned}
formula ::= \quad & \\
& \textbf{\textit{no}}\ expr \\
& \mid \textbf{\textit{lone}}\ expr \\
& \mid \textbf{\textit{one}}\ expr \\
& \mid \textbf{\textit{some}}\ expr \\
& \mid expr \subseteq expr \\
& \mid expr = expr \\
& \mid \neg formula \\
& \mid formula \wedge formula \\
& \mid formula \vee formula \\
& \mid formula \Rightarrow formula \\
& \mid formula \Leftrightarrow formula \\
& \mid \forall\ varDecls \mid formula \\
& \mid \exists\ varDecls \mid formula \\
arity ::= & \ positiveinteger
\end{aligned}
\qquad
\begin{aligned}
expr ::= \quad & \\
& var \\
& \mid expr \cup expr \\
& \mid expr \cap expr \\
& \mid expr \setminus expr \\
& \mid expr\ .\ expr \\
& \mid expr \rightarrow expr \\
& \mid {}^{\sim}expr \\
varDecls ::= & \ var : expr[\,, var : expr]^{*} \\
universe ::= & \ \{atom[, atom]^{*}\} \\
relBound ::= & \ var :_{arity} [constant, constant] \\
constant ::= & \ \{tuple[, tuple]^{*}\} | \{\} [\times \{\}]^{*} \\
tuple ::= & \ \langle atom[, atom]^{*} \rangle \\
atom, var ::= & \ identifier
\end{aligned}
$$

Figure 2.2: Partial Kodkod input syntax taken from Torlak *et al.* [22]

```
1  // ================================================
2    // Kodkod formula:
3  // ================================================
4     // Declarations:
5     no (this/B & this/C) &&
6     (all this: this/B |
7       one (this . this/B.toC) &&
8       (this . this/B.toC) in this/C) &&
9     (this/B.toC . univ) in this/B &&
10    (all this: this/C |
11      (this . this/C.toB) in this/B) &&
12    (this/C.toB . univ) in this/C &&
13    (all this: this/B + this/C |
14      one (this . this/A.id) &&
15      (this . this/A.id) in this/ID) &&
16    (this/A.id . univ) in (this/B + this/C) &&
17
18    // Facts:
19    (all a: this/B + this/C, a': this/B + this/C |
20      !((a . this/A.id) = (a' . this/A.id)) || a = a') &&
21    (all b: this/B, b': this/B | !!(b = b') ||
22      !((b . this/B.toC) = (b' . this/B.toC))) &&
23    (all c: this/C, c': this/C | !!(c = c') ||
24      !(some b: this/B |
25         b in ((c . this/C.toB) & (c' . this/C.toB))))
26 // ================================================
27
```

Figure 2.3: Parts of Kodkod translation pseudo code of the Alloy example

The top part of Figure 2.3 is type, function, and relation declarations, and the bottom part is the translation of the facts. The translation of facts is straight-forward, however, Kodkod has a unique way of declaring types and functions. For example, line 5 states that types `B` and `C` are disjoint, and the domain and range of `toC` are specified in lines 6 to 9. These declarations at the beginning are sufficient for all the type constraints and there is no need to add constraints in translation of facts.

The pseudo code is a comment in the output of the Alloy to Kodkod translation process because Kodkod does not have any input language, and we can only interface to it through Java code. The Java code creates Kodkod structures, and then Kodkod translates them to CNF clauses. Then the SAT solver is called to solve the problem.

Symmetry breaking in Kodkod is done by two methods introduced in Torlak *et al.* [22]. The first method is based on graph automorphism detection, which is complete, and only works reasonably for small problems. The second method is a greedy algorithm, called greedy base partitioning, which finds a subset of the available symmetries that can be detected in polynomial time. This method is not complete, but in practice, works well for medium sized problems. Both methods have poor performance for large problems [22].

2.3 Fortress

Fortress [24] is an SMT-based finite model finder, which uses equality with uninterpreted functions (EUF) as its base logic. EUF is a subset of typed FOL with Equality that excludes quantifiers and adds the equality predicate. Therefore, EUF is decidable. Fortress input is TFOL and a scope for each type. The transformation for flattening the formulas and eliminating the quantifiers is done within Fortress, and the result is translated into the SMT-LIB2 language, and the Z3 SMT solver is called. Figure 2.4 shows the input syntax of Fortress.

The formulas and scopes are passed to Fortress for quantifier expansion. The quantifier expansion fully eliminates the quantifiers expanding them for the finite scope. Fortress' process has four steps [24]. To illustrate these steps, we present an example. Consider the formula

$$\forall x, y : A \bullet \exists z : A \bullet (f(x) = f(z)) \Rightarrow P(y)$$

where the scope of type A is 3.

The first step is to transform formulas into prenex normal form, then skolemize and eliminate existential quantifiers. This means we move all the quantifiers and bound variables to the beginning of the formula, then replace variables bound by the existential

$$
\begin{array}{ll}
\textit{Formulas} & \textit{Terms} \\
\Phi ::= \top \mid \bot \mid p & t : \theta ::= v : \theta \quad \textit{where } v \in V \\
\quad ::= R(t_1 : \theta_1, ..., t_n : \theta_n) & \quad ::= c : \theta \\
\quad ::= \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \Rightarrow \Phi_2 \mid \Phi_1 \Leftrightarrow \Phi_2 & \quad ::= f(t_1 : \theta_1, ..., t_n : \theta_n) \\
\quad ::= \exists v : \theta \bullet \Phi \mid \forall v : \theta \bullet \Phi \quad \textit{where } v \in V &
\end{array}
$$

Figure 2.4: Fortress input syntax taken from Vakili *et al.* [24]

quantifier with constants. After this step it becomes

$$\forall x, y : A \bullet (f(x), f(sk)) \Rightarrow P(y)$$

where the sk is a constant replacing variable z. This constant can be any of the values within the scope of A.

The second step is to generate the universe from the bounds received from the input. Fortress generates constants of the types. In the example, Fortress creates the set of constants $\{a_1, a_2, a_3\}$ in the universe, all of which are of type A and are distinct constants.

The third step is to add range formulas based on the generated universe. These range formulas are added for each function and constant. For our example, the range formulas are

$$sk = a_1 \vee sk = a_2 \vee sk = a3,$$
$$f(a_1) = a_1 \vee f(a_1) = a_2 \vee f(a_1) = a_3,$$
$$f(a_2) = a_1 \vee f(a_2) = a_2 \vee f(a_2) = a_3,$$

and

$$f(a_3) = a_1 \vee f(a_3) = a_2 \vee f(a_3) = a_3.$$

However, the size of the range formulas can be reduced using symmetry breaking techniques. Fortress uses Classen and Sörensson's symmetry breaking technique [7]. In this method, we sort our constants in an arbitrary order, and for each constant occurring in the problem, we pick an element that we have already seen, or a new element, which must be the least unused element in our ordered list. The range formula after symmetry breaking is:

$$sk = a_1,$$

$$f(a_1) = a_1 \vee f(a_1) = a_2,$$
$$f(a_2) = a_1 \vee f(a_2) = a_2 \vee f(a_2) = a_3,$$

and

$$f(a_3) = a_1 \vee f(a_3) = a_2 \vee f(a_3) = a_3.$$

Symmetry breaking reduces the size of the range formulas.

The last step is to ground formulas. At this step, all universal quantifiers are eliminated and all the variables bound by them are replaced by constants. During this process some formulas are simplified if possible. For example, in the formula

$$\forall x, y : A \bullet f(x) \neq f(y) \vee x = y,$$

when both x and y are replaced by a_1, the formula becomes

$$f(a_1) \neq f(a_1) \vee a_1 = a_1,$$

and then it is simplified as $\top$. Also, when x is replaced by a_1 and y is replaced by a_2, the formula becomes

$$f(a_1) \neq f(a_2) \vee a_1 = a_2,$$

and since all the constants generated in the universe are distinct the formula is simplified to

$$f(a_1) \neq f(a_2).$$

Fortress has shown promising results when compared with Kodkod [24]. After fixing some bugs in Fortress, we tested the benchmark again to replicate the result. The comparison result is presented in Table 2.1.

2.4 Z3

Z3 [10] is an efficient SMT solver, which is developed in Microsoft Research Labs. An SMT solver solves Satisfiability Modulo Theories (SMT) problems, which are satisfiability problems for first-order logic with theories. Examples of these theories are: arithmetic, bit-vectors, arrays, and uninterpreted functions. The input language of Z3 is SMT-LIB2 [4].

The algorithm that Z3 uses, is a DPLL-style algorithm [9]. A DPLL algorithm is a SAT algorithm that takes a set of clauses as input and the output is whether the problem is unsatisfiable or a satisfying instance. This DPLL-style algorithm is a heuristic algorithm

Test Results			
Test	Scope	Kodkod (s)	Fortress (s)
Geo158	7	2.00	8.74
Geo158	9	33.37	24.03
Geo158	11	461.39	59.10
Geo091	7	2.45	4.89
Geo091	9	32.03	18.97
Geo091	11	887.49	48.50
Alg212	6	5.67	2.04
Alg212	8	221.18	17.98
Alg212	10	Timed out	803.20
Com008	7	0.45	8.11
Com008	9	0.55	118.98
Com008	11	1.79	538.40
Num374	5	26.25	6.42
Num374	6	304.68	59.29
Num374	7	Timed out	1192.29
Set943	7	4.54	2.50
Set943	9	Timed out	4.51
Set943	11	Timed out	8.37
Set948	7	0.59	2.72
Set948	9	0.88	2.55
Set948	11	1.63	10.34
Med009	7	0.37	57.63
Med009	9	0.38	Memory out
Med009	11	0.44	Timed out
Total time	-	9988.13	6999.56
# of wins	-	11	13
# of timeouts	-	4	2

Table 2.1: Test results for comparison between Kodkod and Fortress (Time limit = 2000s) (In total calculation, Memory out = 2000s, Timed out = 2000s)

that is one of the most efficient algorithms at this time. This algorithm solves a problem by creating a tree and searching for a satisfying assignment, and learning conflict clauses. If a satisfying model is found during the search, Z3 returns the satisfying model. A Z3 satisfying model consists of a specific domain for each type and the functions and predicates are defined over each of these atoms.

SMT-LIB2 is based on TFOL. Each type is disjoint in SMT-LIB2, and therefore, subtyping is not possible by using just the type system. SMT-LIB2 also does not support relations, so for relation and set declarations, we must use predicates for declaration of sets and relations. Unary functions are used to create constants. There are several built-in types, such as numerals, decimals, strings, *etc.* and each of them have their own theory library. The theory libraries are used to find satisfying assignments in clauses that contain expressions in these theories.

2.5 Summary

Since the FOL decision procedure is undecidable, finite model finding approaches can check for a model within relatively small scopes to gain confidence in the model. Fortress and Alloy are two finite model finders. The Alloy Analyzer uses Kodkod as an intermediate language between the Alloy model and SAT solver. Since the test results are in Fortress' favor when compared with Kodkod, we think Alloy models can be solved faster using Fortress. Fortress uses Z3, which is an SMT solver. The input language of Z3 is SMT-LIB2, which is based on TFOL. Z3 uses a DPLL-style algorithm, which is one of the most efficient algorithms at this time.

Chapter 3

Astra

We call the **Alloy to SMT Translator** "Astra", which translates relational FOL to TFOL. In this chapter, we describe our translation process.

3.1 Overview

The Alloy language, uses Kodkod as the intermediate language. For ease of integration with Alloy, we chose Kodkod as the interface, which is much simpler than dealing with all of Alloy's constructs. Our translation requires some reverse engineering since parts of the Alloy model are presented differently in Kodkod.

Our translation starts from a Kodkod file generated by the Alloy tool. Our translation is done in two passes. The first pass is to determine the types, relations, and functions declarations. In this pass, Astra extracts and declares all the relations, their types, their arities, and their multiplicities. During the second pass, the formula is translated using the information extracted in the first pass.

There are different equivalent translations for each model. We call each of the independent factors an axis. There are three different axes:

- Type declaration axis: Typed, Untyped.
- Function declaration axis: Functions, Predicates.
- Scope translation axis: Fortress, SMT finite model finding (SMT FMF), Unscoped.

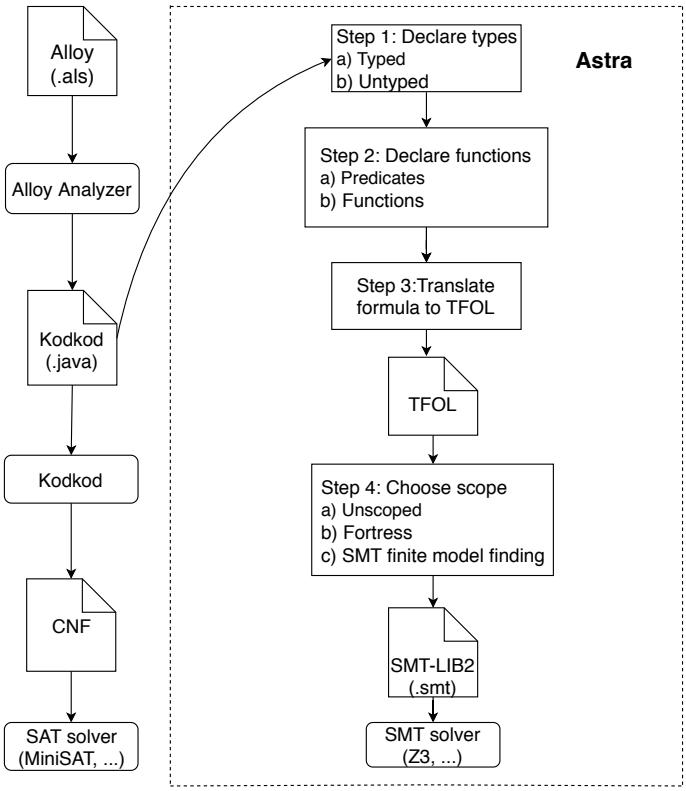

Figure 3.1: Steps in translation.

Figure 3.1 presents an outline of the steps we use to translate Kodkod to TFOL. TFOL supports only total functions. Our goal is to create a very direct translation to TFOL from Alloy. Next, we overview the steps in our translation and the options we investigate for each step. We explain these steps in detail with the following subsections.

Step 1: Declaring Types. Alloy uses sets as types. In the Kodkod representation, there is one universe of elements and unary relations constrain the elements to be of a certain type when needed. In TFOL, we have the ability to use its type space to separate the elements into types and thereby reduce the size of the set of elements for each quantifier. *We hypothesize that using types will improve the performance of the SMT solver.* Our first translation step is to declare types in TFOL. We explore two options: untyped and typed.

Step 2: Declaring Functions. The next step of translation is to declare the total functions in TFOL. A predicate in TFOL is a function that returns a Boolean value. *We hypothesize that solving a problem with theories is harder than solving a SAT problem in an SMT solver. Therefore, using predicates instead of functions can improve the performance,*

at least for UNSAT problems, since if the SMT solver cannot find a satisfying Boolean assignment, then it does not have to use its theories. Thus, we implement two options for declaring TFOL functions. The first, which we call "predicates", is to make all Kodkod relations predicates in TFOL. The second, which we call "functions", is to use as many non-predicate functions as possible in TFOL.

Step 3: Translating Formulas. Step 3 is to translate the formulas in the Kodkod datatype to formulas in TFOL. In this step, we have to translate set operations (union, intersection, *etc.*) into their equivalent in FOL.

Step 4: Choosing Scopes. In Step 4, we transform the formula into prenex normal form and set the scope for the problem. *We hypothesize that if the problem is within a decidable fragment of TFOL, the SMT solver can solve it quickly without any bounds since the formula is smaller without introduced constants and their expansion.* We also want to investigate whether quantifier expansion with finite scopes is more efficiently done by the SMT solver or prior to solving. We investigate three methods for handling the scope: Fortress, SMT FMF, Unscoped.

Step 5: SMT Solver. In the final step, Step 5, the SMT solver is called on the problem.

We have not yet proven the correctness of our translation, however, since the translation is from Kodkod (rather than Alloy) the number of cases to consider for correctness is reduced and can be seen more directly.

In the following, the translation process, its options, and the combination of the options is explained thoroughly. For illustration of the translations, the example in Figure 2.1 is used.

3.2 Step 1: Declaring SMT types

Since Kodkod does not have a type system, it considers everything to be of the same type, and the different types of the Alloy signatures are unary relations. When translating Kodkod to SMT, types can play a significant role in the analysis performance of the tool because SMT solvers use typed FOL. Astra finds the Alloy types by doing a thorough search in the whole formula. Kodkod type names are auto-generated and may differ from the original names in Alloy so they can remain unique as the Alloy structures are merged together.

In Astra, the type derivation process is done in the first pass. In this pass, most of the formulas are not interesting, and a recursive process collects only the parts of the

formula that are about types and functions/relations. For n-ary relations in Kodkod, the multiplicities, which indicate whether the relation is a total function, partial function, or just a relation, are detected. In this step, if the typed option is chosen, then upon detection of inheritance, the supertype is split into the subtypes, and only the leaves of the type hierarchy is declared as SMT types. For the untyped option on the other hand, everything is assumed to be of the universal type, and all of the leaf types in type hierarchy are translated as predicates.

For the example of Figure 2.1, Table 3.1 and 3.2 show the state of the translator after this step for the typed and untyped options respectively. At this stage, types are declared, but the types of the functions are not detected yet. Hence, functions cannot be declared in this step.

Types	**Relation arity**	**Relation multiplicity**
`B`	(“`id`”, 2)	(“`id`”, “`one`”)
`C`	(“`toC`”, 2)	(“`toC`”, “`one`”)
`ID`	(“`toB`”, 2)	(“`toB`”, “`set`”)

Table 3.1: Types, functions and relations after detection in pass 1 for typed option.

Types	**Relation arity**	**Relation multiplicity**
Univ	(“`id`”, 2)	(“`id`”, “`one`”)
	(“`toC`”, 2)	(“`toC`”, “`one`”)
	(“`toB`”, 2)	(“`toB`”, “`set`”)
	(“`B`”, 1)	(“`B`”, “`set`”)
	(“`C`”, 1)	(“`C`”, “`set`”)
	(“`ID`”, 1)	(“`ID`”, “`set`”)

Table 3.2: Types, functions and relations after detection in pass 1 for untyped option.

3.3 Step 2: Declaring SMT functions and relations

After we have declared the types and found the name, arities, and multiplicities of the relations in Step 1, we have different options for how to declare functions in TFOL. Next

we explain the function and relation declarations for each of the typing options of Step 1.

By using the "bounds" structure of Kodkod the functions' and relations' types can be determined when using the typed option of Step 1. For non-hierarchical types, this process is straight-forward, because all the unary relations are mapped to exactly one SMT type. For hierarchical types, some functions and relations may have been defined over several types, since the domain or range of the relation can be a supertype. In this case, since all the types in SMT are disjoint, we create different functions with only the leaf types involved for the functions. In general, a function such as

$$f : Super_1 \rightarrow Super_2$$

with their domain or range being supertypes, is split into multiple functions:

$$f_{11} : Sub1_1 \rightarrow Sub2_1, ..., f_{mn} : Sub1_m \rightarrow Sub2_n.$$

For example, for the function `id` in the Figure 2.1 Astra detects that the function is defined over two types, `B` and `C`. Table 3.3 shows the function and predicate declarations after Step 2.

Declarations with functions	**Declarations with predicates**
$id_B : B \mapsto ID$	$id_B : B \times ID \mapsto Bool$
$id_C : C \mapsto ID$	$id_C : C \times ID \mapsto Bool$
$toC : B \mapsto C$	$toC : B \times C \mapsto Bool$
$toB : C \times B \mapsto Bool$	$toB : C \times B \mapsto Bool$

Table 3.3: Function and predicate declarations after the pass 1 for typed option

The function declaration step has two options: Functions, Predicates. We choose the functions option when we want the functions to be translated as SMT functions. But we can also translate them as predicates, with additional constraints. The multiplicity constraints are:

- `one` (total function):

$$\forall a : A \bullet \exists b : B \bullet R(a, b) \wedge \forall b' : B \bullet R(a, b') \Rightarrow b = b'$$

- `lone` (partial function):

$$\forall a : A \bullet \forall b, b' : B \bullet R(a, b) \wedge R(a, b') \Rightarrow b = b'$$

- `some` (relation with each member of the domain mapped to at least one member of the range):

$$\forall a : A \bullet \exists b : B \bullet R(a, b)$$

- `set` (relation):

$$None$$

Hence, the constraints for the example are:

$$\begin{aligned} &\forall x : B \bullet \exists y : ID \bullet id_B(x, y) \wedge \forall x' : B \bullet id_B(x', y) \Rightarrow x = x' \\ \wedge\ &\forall x : C \bullet \exists y : ID \bullet id_C(x, y) \wedge \forall x' : C \bullet id_C(x', y) \Rightarrow x = x' \\ \wedge\ &\forall x : B \bullet \exists y : C \bullet toC(x, y) \wedge \forall x' : B \bullet toC(x', y) \Rightarrow x = x' \end{aligned}$$

After this step, the function declarations are complete.

The untyped option of Step 1 does not include types and there is only one universal type, $Univ$, so in Step 2 the declarations depend only on the multiplicity and the arity. The general method is similar to the typed option of Step 1. The functions and predicates are created are shown in Table 3.4. For the example of Figure 2.1 the following formulas are added to the translation to constrain declared relations to be functions if we choose the predicates option in Step 2.

$$\begin{aligned} &\forall x : Univ \bullet \exists y : Univ \bullet id(x, y) \wedge \forall x' : Univ \bullet id(x', y) \Rightarrow x = x' \\ \wedge\ &\forall x : Univ \bullet \exists y : Univ \bullet toC(x, y) \wedge \forall x' : Univ \bullet toC(x', y) \Rightarrow x = x' \end{aligned}$$

One of the notable differences between Table 3.3 and 3.4 is that there are two `id` functions in the former, and one in the latter, and each Alloy type has a membership predicate. The declaration of only one `id` function in Table 3.4 is because when the untyped option is chosen, type constraints are imposed with the type predicates, and therefore, there is no need to separate subtypes `B` and `C`.

3.4 Step 3: Formula translation

In this section, we present the translation options from relational FOL terms to TFOL terms. To create as direct a translation to TFOL as possible, we represent each set operation using the characteristic predicate for the set and the propositional operation that is

Declarations with functions	Declarations with predicates
$id : Univ \mapsto Univ$	$id : Univ \times Univ \mapsto Bool$
$toC : Univ \mapsto Univ$	$toC : Univ \times Univ \mapsto Bool$
$toB : Univ \times Univ \mapsto Bool$	$toB : Univ \times Univ \mapsto Bool$
$B : Univ \mapsto Bool$	$B : Univ \mapsto Bool$
$C : Univ \mapsto Bool$	$C : Univ \mapsto Bool$
$ID : Univ \mapsto Bool$	$ID : Univ \mapsto Bool$

Table 3.4: Function and predicate declarations after the pass 1 for untyped option

the equivalent of the set operation. For example, the union of two sets is the disjunction of the characteristic predicate for each of the operands to the union.

This translation can be done in either a top-down or bottom-up traversal of the Kodkod formula data structure. To handle the generality of set expressions in Alloy, and ease of later development for transitive closure and simplification methods, we choose a bottom-up traversal to facilitate its compositionality and so that the types of terms can be determined from their leaves on the way up. To make a bottom-up traversal possible, we have to provide a translation for each term, not just each formula. Our translation is defined by the $[\cdot]$ operator, which takes a term in Kodkod, and translates it into TFOL term. We define $[\cdot]$ in this section. The type of the Kodkod term must be determined as we walk up the data structure; we use the notation $\textsc{type}(\cdot)$ to denote this calculation. Figure 3.2 and 3.3 represent the complete translations for typed and untyped options.

The leaves of the Kodkod formula datatype are variables or relations of certain types. We need this type information in our translation to TFOL. Kodkod stores the types of its variables with the variable, so the translation of a Kodkod variable or relation is simply a term of the type in the Kodkod, as in $[v : t]$ in bottom part of the Figure 3.2 and 3.3.

Next, we describe the translation for the set operations. A term in Kodkod must return a term in TFOL so we translate each non-leaf term into a helper relation and add a constraint for the meaning of the helper relation. For example, $[A \cup B]$ where A and B are both of type t is a new relation R of type $t \rightarrow Bool$ with an additional constraint of $\forall x : t \ \bullet \ R(x) \Leftrightarrow A(x) \vee B(x)$. Thus, the translation of the set operations results in a TFOL term plus declarations and additional constraints. In the "Set operations" part of the Figure 3.2 and 3.3, each R_{new} is the new relation name.

By using this method of helper relations, our translation results in possibly smaller clauses but more of them compared to the methods that use top-down approach. This

Propositional connectives :		
$[true]$	$:=$	$true$
$[false]$	$:=$	$false$
$[\neg A]$	$:=$	$\neg[A]$
$[A \wedge B]$	$:=$	$[A] \wedge [B]$
$[A \vee B]$	$:=$	$[A] \vee [B]$
$[A \Rightarrow B]$	$:=$	$[A] \Rightarrow [B]$
$[A \Leftrightarrow B]$	$:=$	$[A] \Leftrightarrow [B]$
Quantified formulas :		
$[\forall(x : t) \bullet A]$	$:=$	$\forall x : t \bullet [A]$
$[\exists(x : t) \bullet A]$	$:=$	$\exists x : t \bullet [A]$
Set operations :		
$[A \subseteq B]$	$:=$	$\forall x : \text{TYPE}(A) \bullet [A](x) \Rightarrow [B](x)$
$[(v : t) \in B]$	$:=$	$[B](v : t)$
$[A \cup B]$	$:=$	$R_{new} : \text{TYPE}(A) \rightarrow Bool$
		add $\forall x : \text{TYPE}(A) \bullet R_{new}(x) \Leftrightarrow [A](x) \vee [B](x)$
$[A \cap B]$	$:=$	$R_{new} : \text{TYPE}(A) \rightarrow Bool$
		add $\forall x : \text{TYPE}(A) \bullet R_{new}(x) \Leftrightarrow [A](x) \wedge [B](x)$
$[A - B]$	$:=$	$R_{new} : \text{TYPE}(A) \rightarrow Bool$
		add $\forall x : \text{TYPE}(A) \bullet R_{new}(x) \Leftrightarrow [A](x) \wedge \neg[B](x)$
$[^{\sim}A]$	$:=$	$R_{new} : \text{TYPE}(ran(A)) \times \text{TYPE}(dom(A)) \rightarrow Bool$
		add $\forall x : \text{TYPE}(ran(A)), y : \text{TYPE}(dom(A)) \bullet$
		$\quad R_{new}(x, y) \Leftrightarrow [A](y, x)$
$[A.B]$	$:=$	$R_{new} : \text{TYPE}(dom(A)) \times \text{TYPE}(ran(B)) \rightarrow Bool$
		add $\forall x : \text{TYPE}(dom(A)), y : \text{TYPE}(ran(B)) \bullet$
		$\quad R_{new}(x, y) \Leftrightarrow \exists z : \text{TYPE}(ran(A)) \bullet [A](x, z) \wedge [B](z, y)$
$[A = B]$	$:=$	$[A] = [B]$
$[v : t]$	$:=$	$v : t$
$[((v_1 : t_1), (v_2 : t_2))]$	$:=$	$[v_1] \times [v_2]$

Figure 3.2: Formula translation for typed option.

method reduces the size of formulas after quantifier expansion for finite model finding.

For the functions option of Step 2, there are special cases for join where the first argument, v, to join is a variable or the first argument is the result of a total function application (which results in a scalar), we translate this expression to function application:

$$\begin{array}{rcl} [(v:t).f] & := & (f(v:t)) : \textsc{type}(ran(f)) \\ [(v:t).f_1.f_2] & := & (f_2(f_1(v:t))) : \textsc{type}(ran(f_2)) \end{array}$$

When translating quantified formulas, we check the type of the quantified variables. If a variable of a supertype is detected, then the formula is split into all the subtypes of the supertype. In general, splitting a quantified formula over supertypes such as

$$\forall x : Super \bullet \Phi \quad and$$

$$\exists x : Super \bullet \Phi,$$

into subtypes results in

$$(\forall x_1 : Sub_1 \bullet \Phi) \wedge ... \wedge (\forall x_n : Sub_n \bullet \Phi) \quad and$$

$$(\exists x_1 : Sub_1 \bullet \Phi) \vee ... \vee (\exists x_n : Sub_n \bullet \Phi)$$

respectively.

For example, the `uniqueID` fact in the example of Figure 2.1, is translated as

$$(\forall b, b' : B \bullet id_B(b) = id_B(b') \Rightarrow b = b') \wedge (\forall c, c' : C \bullet id_C(c) = id_C(c') \Rightarrow c = c')),$$

and fact `uniqueB` is translated as

$$\forall b, b' : B \bullet \neg(b = b') \Rightarrow \neg(toC(b) = toC(b')),$$

and fact `uniqueC` is translated as

$$\forall c, c' : C \bullet \neg(c = c') \Rightarrow \neg(\exists b : B \bullet R_{new}(b)),$$

where R_{new} is a new relation of type $B \rightarrow Bool$ and

$$\forall b : B \bullet R_{new}(c) \Leftrightarrow toB(c, b) \wedge toB(c', b)$$

is added to the end of the translated formula for `uniqueC`.

If the untyped option is chosen in Step 1, this translation changes slightly. The types all become the universal type and unary predicates are added as appropriate to limit the formula to the correct type. For example, the translation for the union operator is:

Propositional connections :		
$[true]$	$:=$	$true$
$[false]$	$:=$	$false$
$[\neg A]$	$:=$	$\neg[A]$
$[A \wedge B]$	$:=$	$[A] \wedge [B]$
$[A \vee B]$	$:=$	$[A] \vee [B]$
$[A \Rightarrow B]$	$:=$	$[A] \Rightarrow [B]$
$[A \Leftrightarrow B]$	$:=$	$[A] \Leftrightarrow [B]$
Quantified formulas :		
$[\forall(x : t) \bullet A]$	$:=$	$\forall x : Univ \bullet t(x) \Rightarrow [A]$
$[\exists(x : t) \bullet A]$	$:=$	$\exists x : Univ \bullet t(x) \wedge [A]$
Set operations :		
$[A \subseteq B]$	$:=$	$\forall x : Univ \bullet [A](x) \Rightarrow [B](x)$
$[(v : t) \in B]$	$:=$	$[B](v : Univ)$
$[A \cup B]$	$:=$	$R_{new} : Univ \to Bool$
		add $\forall x : Univ \bullet P_{\mathrm{TYPE}(A)}(x) \Rightarrow R_{new}(x) \Leftrightarrow [A](x) \vee [B](x)$
$[A \cap B]$	$:=$	$R_{new} : Univ \to Bool$
		add $\forall x : Univ \bullet P_{\mathrm{TYPE}(A)}(x) \Rightarrow R_{new}(x) \Leftrightarrow [A](x) \wedge [B](x)$
$[A - B]$	$:=$	$R_{new} : Univ \to Bool$
		add $\forall x : Univ \bullet P_{\mathrm{TYPE}(A)}(x) \Rightarrow R_{new}(x) \Leftrightarrow [A](x) \wedge \neg[B](x)$
$[\sim A]$	$:=$	$R_{new} : Univ \times Univ \to Bool$
		add $\forall x : Univ, y : Univ \bullet$
		$P_{\mathrm{TYPE}(ran(A))}(x) \wedge P_{\mathrm{TYPE}(dom(A))}(y) \Rightarrow$
		$R_{new}(x, y) \Leftrightarrow [A](y, x)$
$[A.B]$	$:=$	$R_{new} : Univ \times Univ \to Bool$
		add $\forall x : Univ, y : Univ \bullet$
		$P_{\mathrm{TYPE}(dom(A))}(x) \wedge P_{\mathrm{TYPE}(ran(B))}(y) \Rightarrow$
		$R_{new}(x, y) \Leftrightarrow \exists z : Univ \bullet$
		$P_{\mathrm{TYPE}(ran(A))}(x) \wedge [A](x, z) \wedge [B](z, y)$
$[A = B]$	$:=$	$[A] = [B]$
$[v : t]$	$:=$	$v : Univ$
$[((v_1 : t_1), (v_2 : t_2))]$	$:=$	$[v_1 : t_1] \times [v_2 : t_2]$

Figure 3.3: Formula translation for untyped option.

$$[A \cup B] \;:=\; R_{new} : Univ \to Bool \\ \text{add } \forall x \bullet P_{\text{TYPE}(A)}(x) \Rightarrow (R_{new}(x) \Leftrightarrow [A](x) \vee [B](x))$$

where $P_{\text{TYPE}(A)}(x)$ is the predicate for the type of set A. The change also affects the translation of the multiplicity constraints in Step 2 in a similar manner.

To compare both type option translations' effect on inheritance, the fact `uniqueID` in the untyped setting is translated as

$$\forall x, x' : Univ \bullet (B(x) \wedge B(x')) \vee (C(x) \wedge C(x')) \Rightarrow id(x) = id(x') \Rightarrow x = x'.$$

In this translation, the formula is shorter because it is not split for all subtypes of type A.

The alternative of a top-down traversal would have been more difficult to correctly implement. It would result in longer formulas, but no extra quantified constraints. In a top-down traversal of a nested set expression, variables of unknown type would have had to be created on the way down the traversal so that formulas would always be passed back up the traversal.

3.5 Step 4: Scope translation

The scope of the types is extracted from the Kodkod list of atoms (atomlist). Our translator traverses this list and maps each type to its scope.

Fortress. If the Fortress scope option is chosen, then the scope is passed to Fortress. Fortress creates an SMT-LIB2 file and calls Z3. In this method, the end result is a large quantifier-free formulas in EUF.

SMT FMF. The second scope translation option, which we call SMT finite model finding (SMT FMF), is to just add the range formulas to the formula. This process differs depending on the type translation option of Step 1 chosen. If the type translation is typed, then the range formula is added as the third step in the quantifier expansion process in Fortress.

For untyped option of Step 1 where there is only one type ($Univ$), the usual range formulas are not effective because we want to limit the number of values on each type rather than the total number of values of the $Univ$ type. Thus, the constraints are added as a part of a formula. These constraints set the minimum number of each type, which is equal to the values specified in the Alloy model. Then Fortress sets the exact number of all atoms, which is the size of the $Univ$ type. These two constraints together set the exact

scope for the problem. In general, the untyped range formula in a model with m types each with n_i atoms is

$$\forall x : Univ \bullet (A_1(x) \Rightarrow \neg A_2(x) \wedge ... \wedge \neg A_m(x)) \wedge ...$$
$$\wedge (A_m(x) \Rightarrow \neg A_1(x) \wedge ... \wedge A_{m-1}(x)),$$
$$\bigwedge_{i=1}^{m} (\exists x_1, ..., x_{ni} \bullet A_i(x_1) \wedge ... \wedge A_i(x_{ni}) \wedge x_1 \neq x_2 \wedge ... \wedge x1 \neq x_{ni} \wedge ...$$
$$\wedge x_{ni-1} \neq x_{ni}).$$

The range formula for the untyped option in the example of Figure 2.1 is

$$\forall x : Univ \bullet B(x) \Rightarrow \neg C(x) \wedge C(x) \Rightarrow \neg B(x),$$
$$\exists x_1, x_2, x_3 : Univ \bullet B(x_1) \wedge B(x_2) \wedge B(x_3) \wedge x_1 \neq x_2 \wedge x_2 \neq x_3 \wedge x_1 \neq x_3,$$

and

$$\exists x_1, x_2, x_3 : Univ \bullet C(x_1) \wedge C(x_2) \wedge C(x_3) \wedge x_1 \neq x_2 \wedge x_2 \neq x_3 \wedge x_1 \neq x_3.$$

3.5.1 Unscoped

The final option for scope translation is to ignore the scope and pass the formula without limited scopes. In this option the SMT solver searches for a solution that may be of infinite scope. If a problem is found to be UNSAT for unbounded scope, then the result is UNSAT for any given scope. Also, the decision procedure in this option is an undecidable problem that might not terminate.

3.6 Step 5: SMT solver

At this stage, every data structure is translated into SMT-LIB2 and a file is created. Then the SMT solver is called for solving the problem, and the result is reported to the user.

3.7 Implementation

Astra is implemented in Java as a solver that takes Kodkod's data structures as arguments to facilitate easy future integration with the Alloy Analyzer. We use the Fortress library implemented by Vakili and Day [24] as our abstract datatype for TFOL declarations and formulas.

3.8 Related work

In this section we discuss and compare similar translations to ours.

With respect to type declarations, KeY [6] types are not required to be disjoint, therefore the inheritance translation is straight-forward in work of Ulbrich *et al.* [23]. In El Ghazi *et al.* [12], the inheritance translation is a mixture of our typed and untyped translation options. Every supertype is translated into a disjoint type in SMT, and then the subtypes are translated as membership predicates. The rest of the type translations are quite similar to our work.

With respect to our function declaration option, in Ulbrich *et al.* [23], the approach for translating different types of relations and functions is very similar to ours by creating a helper relation and adding extra constraints. The declaration of different types of functions and relations in El Ghazi *et al.* [12] requires definition of additional relations and functions and constraining them with additional constraints. For example, to define a partial function, El Ghazi *et al.* [12] create a relation, and a function and then add a constraint on the relation that every member of the relation is a member of the function.

With respect to formula translation, our translation is similar to what has been done in Ulbrich *et al.* [23] and El Ghazi *et al.* [12]. The difference is that their translation approach is top-down. The top-down translation may results in shorter formulas, with less relations. But using the bottom-up approach, the combinations of the relations and functions can be stored and reused. Hence, in some models, the bottom-up approach can result in shorter formula. In addition, the bottom-up approach can facilitate the future implementation of transitive closure.

With respect to the scope translation options, both Ulbrich *et al.* [23] and El Ghazi *et al.* [12] solve problems for unbounded scopes, while we provide different scope translation options. This way, for the problems that cannot be solved for the unbounded scope, we can use finite model finding approaches.

Overall, our focus is to find the best translation combination for solving different sets of problems. Other similar work focuses on only one combination and testing the performance for that one combination. While our result shows a performance advantage in most of our test cases, we also provide guidelines for improving modelling process, as well as existing tools for solving the models.

3.9 Summary

We presented our translation process and explained each step thoroughly. Also, we proposed three hypotheses regarding modelling and performance results. Then we discussed different translation options, and illustrated them by examples. We explained our implementation, and compared our translation to related work.

Chapter 4

Performance Testing

In this chapter, we discuss the results of our performance tests. We examine our hypotheses and explain the results. Finally, we consider the results overall and present questions and discussion regarding the unexpected results that we observe. Each test case has different characteristics. The characteristics are chosen from different aspects of a model that we hypothesize can affect the performance time. All of these tests are compared and presented against each other to better illustrate the differences between different combinations and Kodkod.

4.1 Astra options

Our test cases are used to test all of the different options of Astra, to extract practical data to the extent that conclusions can be drawn from them. Not all different combinations of the different axes are efficient and interesting in terms of performance time. Some, unexpectedly, are eliminated due to poor practical test results, and some we expected to show poor results. For example, we expect functions to make a formula grow more slowly in size than relations, and therefore, choosing the Fortress option with predicates, is expected to have poor results relatively. We look deeper into the interesting combinations. The complete set of combinations and whether they are considered interesting in this thesis or not is illustrated in Table 4.1. The reasons some of the combinations are rejected are explained in the following (D stands for discussion):

- D1: Fortress with functions is a good combination, because the slower growth of the functions with scope growth, compared to predicates, reduces the quantifier

Table 4.1: Translation combinations. ✓ = interesting combination, ✗ = rejected combination

	Functions			**Predicates**		
	Unscoped	Fortress	SMT FMF	Unscoped	Fortress	SMT FMF
Typed	✗: D4	✓: D1, D3	✗: D4	✓: D3, D5	✗: D2	✗: D6
Untyped	✗: D4	✗: D6	✗: D4	✓: D5	✗: D2	✓: D5, D6

expansion time. So we expected Fortress with functions option would have good performance and it did.

- D2: We expected the combination of predicates and Fortress to result in poor performance since the formula grows larger with predicates rather than functions as the scope of the problem grows. These combinations did have poor performance.
- D3: We expected the SMT type system would help combinations with the typed option to perform well. These combinations did have good performance.
- D4: We expected the combination of functions and SMT FMF or unscoped would have bad performance because less usage of theories in models with quantified formulas can cause improvement to the performance. These combinations did have poor performance.
- D5: We expected the combination of predicates and SMT FMF or unscoped to be a good combination because the usage of predicates instead of functions in UNSAT models with quantified formulas can result in less usage of theories. These combinations did have good performance in three of four combinations.
- D6: Good or poor performance is merely based on the empirical data, and we do not have an explanation.

4.2 Characteristics of the models

In this section, we discuss the characteristics of the test cases. These test cases come mainly from the Fortress benchmark [24] that is taken from the Kodkod benchmark [22]. We created additional tests to have a more diverse benchmark for Astra. There are in total eleven Alloy models and they are tested for different scopes, making 29 tests in total.

Each test has certain characteristics that affects the tool's performance. We propose a set of parameters that we hypothesize can affect the performance. We measure the number of types, the depth of the function applications, the length of the joins, the depth of quantifiers, the arity of the relations and functions, the number of functions, and the number of relations into account.

The **number of types** is the number of distinct signatures declared in Alloy. This can affect performance since the existence of types in a model means extra constraints on the model. The effect of types can be examined through the translation options, by setting the option to typed or untyped for one problem. It can also show its effect in different test cases with different number of types.

The **depth of the function applications** is a parameter that represents the maximum nestedness of function application in the Alloy model. Nested function applications result in long joins, since function applications are translated as joins in Kodkod, but they will be translated as function applications in Astra as we detect them by reverse engineering. We hypothesize this detection will improve the performance significantly, because the join operation causes creation of new relations and long joins can result in creation of new helper relations with impractical arity that makes the analysis practically impossible.

The **length of the joins** for non-function relations or for the predicates option can result in creation of new helper relations with large arities that can affect performance significantly.

The **depth of the quantifiers** specify the number of quantified variables in the model after the model is transformed into prenex normal form. The depth of universal and existential quantifiers are measured separately, as their effect on the performance may vary. In some combinations, the depth of the quantifiers is increased due to adding range formulas or other helper formulas, however, these helper variables are not part of the original model and are not counted.

The **arity of the relations and functions** is one of the most important parameters because it causes exponential growth with the scope of the problem. For this parameter, the maximum arity presented in the model is measured.

Lastly, the **number of functions and relations** in the model, separately, are two parameters that affects the performance. Especially, in the options with quantifier expansions, as the number of functions and relations increases, the performance time increases to the point that the quantifier expansion becomes impractical.

We hypothesize that each of these parameters play a role in the analysis performance of the tool for a model. We measure and record these characteristics, and compare each

Table 4.2: Characteristics of the tests.(* = New test)

Test case characteristics								
Test	# T	Application	Join length	Forall	Exists	Arity	# F	# R
Num347	1	4	18	3	7	3	3	0
Top020*	1	2	3	4	10	3	5	5
Com008	2	0	1	3	3	2	0	3
Infinity*	1	1	2	2	2	1	1	0
Set943	1	2	4	4	5	3	2	3
Set948	1	2	4	4	7	4	3	4
Alg212	1	2	12	4	5	4	1	0
Geo091	1	1	2	5	10	3	1	5
Geo158	1	1	2	5	8	3	1	5
Med009	1	0	1	3	4	2	0	19
GraphColoring*	3	1	1	2	1	1	2	1

model and the effect of them on the performance time and evaluate the results, so that we can have a quantified way of measuring a problem's difficulty for the analysis tools. Also, as there are various equivalent ways to translate a model, guidelines can be proposed based on the effect of each parameters, so that modellers can create more efficient models and choose the appropriate analysis combination.

4.3 Case studies

We chose a number of Alloy models that cover a range of interesting characteristics: 1) number of types (# T); 2) maximum depth of applications (Application); 3) maximum number of connected joins (join length); 4) maximum nesting of universal quantifiers (forall); 5) maximum nesting of existential quantifiers (exists); 6) maximum arity of a relation (arity); 7) number of total functions (# F); and 8) number of relations (# R). We manually measured these characteristics for each of our models. Table 4.2 represents all the raw data values. Figure 4.1 uses a radar chart to illustrate how our different models covered these characteristics. The values are normalized to 5.

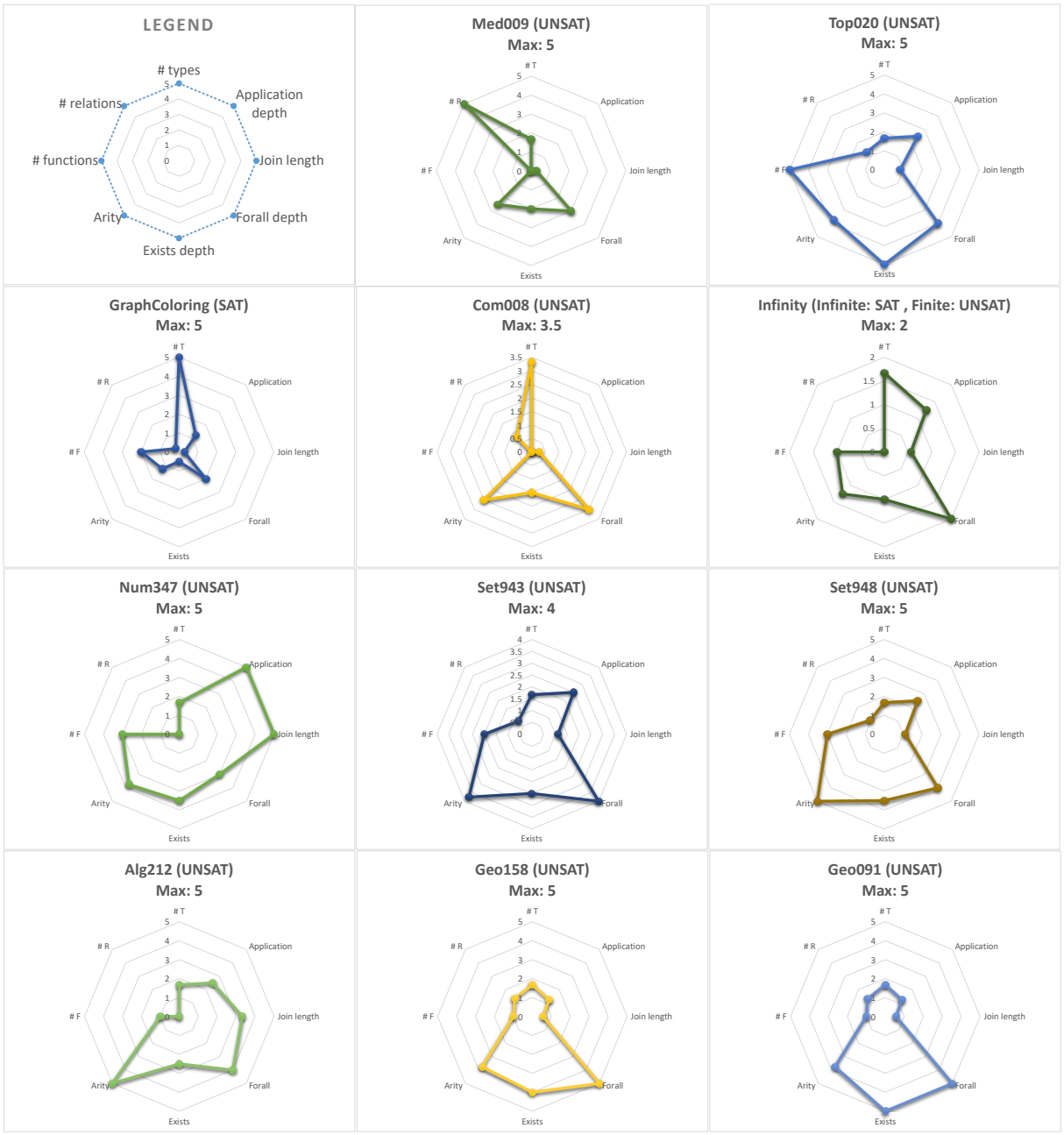

Figure 4.1: Characteristics of tests.

4.4 Performance results

All of our tests were completed on a computer with 2.6-GHz Intel Core i5 CPU, with a 2500MB Memory limit and a 2000 second time limit for each process. Figure 4.2 shows our results using a logarithmic scale for time on the y-axis with the twenty-nine tests across the x-axis. The times include the time for translation and the time for SMT solving (or SAT solving in Kodkod). The five option combinations with interesting results are shown by different lines in the graph. The lowest line for a test on the graph means the best performance. Any lines that hit the uppermost point on the graph mean that we stopped the test after it had taken 2000 seconds or had run out of memory.

The differences in the performance of the methods is quite considerable in most tests, ranging from less than a second for the best performing method to over 2000 seconds for the worst performing method.

In nine of the eleven models, one of our translation combinations produces better results than Kodkod. The typed, predicates, unscoped option had the best performance in five models and tied for the best performance on three other models making it the clear winner in performance. Every combination of options that we included won at least one test. Overall, the combination of typed, functions, Fortress performed the best for the scoped combinations (including Kodkod).

4.4.1 Statistical model

In this subsection, we try to find correlation between model characteristics and translation combinations based on our test results.

Although we do not have a large benchmark, we tried the linear regression method to find a correlation between the model characteristics and the performance of a method. Since the number of tests is relatively small for such method, we used it only to rank the characteristics, and we present a model only for the ones with an R-squared larger than 0.6. This threshold eliminated the models for Kodkod and the untyped, predicates, SMT FMF method.

The first statistical model we present is for the option combination of typed, functions, Fortress combination. This statistical model's R-squared is 0.76, and it points out that the number of types, the depth of function applications, and the number of relations, play the most important roles in its performance.

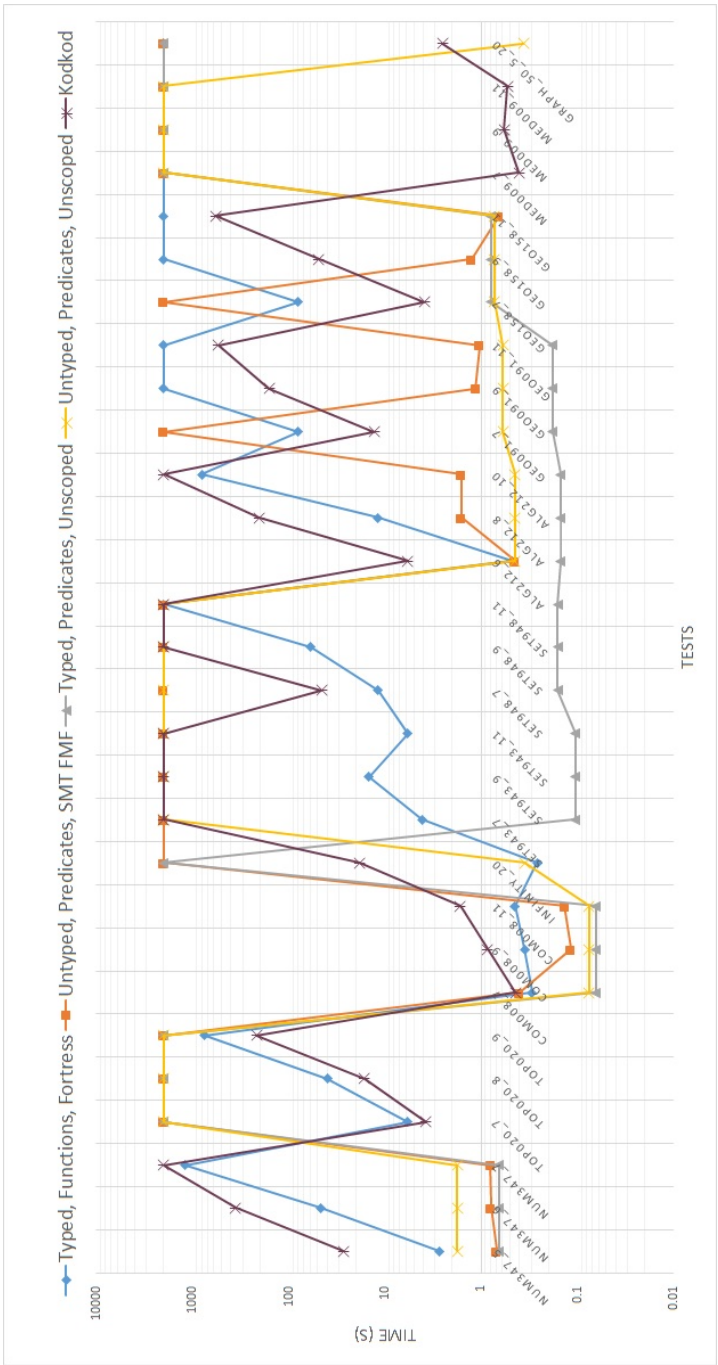

Figure 4.2: Performance results for tests.

The second statistical model is for the typed, predicates, unscoped combination, which has an R-squared of 0.65. This statistical model points out that the number of types and the number of functions and relations play the biggest role in performance time of the tool.

Lastly, the untyped, predicates, unscoped combination has the best suiting statistical model with an R-squared of 0.86. This statistical model, as expected, does not value the number of types as much. This statistical model mostly emphasizes the arity of the functions and relations, the depth of function applications, and the number of functions and relations.

These results give us insight about which options may be more suitable for problems of certain characteristics. For example, we observed that SAT tests with many types are better solved by the untyped option, while tests with functions of large arities may be better solved with other options. These insights can be used as guidelines in the Alloy modelling process to create models that can be solved more easily by a specific option.

4.5 Evaluation

In this section we evaluate the results, discuss our hypotheses, explain the correlations between model characteristics and translation combinations, present guidelines for modelling and analysis, and finally, discuss the results that we could not explain.

4.5.1 Hypotheses

In this subsection, we discuss our hypotheses based on our test results.

- *H1: We hypothesized that using types would improve the performance of the SMT solver.*

  When working with the Fortress option, the typed option works much better than untyped. However, although there are slightly better results with the typed option for other option combinations, the results are not conclusive.

- *H2: We hypothesized that using predicates instead of functions, can improve performance, at least for UNSAT problems.*

  Most of the UNSAT problems are solved very quickly when predicates are used, however, they timed out, or ran out of memory, when the functions option is used.

- *H3: We hypothesized that the unscoped option would often be faster than finite scopes.*

  Many of the tests with unscoped option are solved within seconds. However, it is unclear whether this good performance is due to the unscoped option or the predicates option.

- *H4: We hypothesized that our set of model characteristics affect performance results.*

  Based on the results, the depth of quantifiers is not as important as we hypothesized. Also, we found out the number of relations and functions, depth of function and relation application, and function and relation arities affect the performance the most in all combinations.

4.5.2 Guidelines for suitable analysis combination

In this subsection, we provide some guidelines regarding modelling and model analysis based on the results we described above. We propose a decision tree based on the empirical data to decide which combination to choose for a problem with certain characteristics. The decision tree is shown in Figure 4.3. Based on the empirical results, we find the typed, predicates, unscoped combination to be the most successful combination. This combination is chosen as the default. However, if the the problem has a large number of relations, then Kodkod is the most successful option. If the number of functions in a model is large, then both Kodkod and the untyped, predicates, unscoped combination have shown promising results. Lastly, for SAT models, it is better to use either the typed, functions, Fortress combination or the untyped, predicates, unscoped combination. In summary, our guidelines are:

- G1: Models with large function arities are best solved by types, predicates, unscoped combination, based on statistical models.
- G2: For unbounded scope modelling in SMT solvers, for a better chance of solving UNSAT problems, functions must be declared as predicates with extra constraints, based on H1 and empirical evidence.
- G3: For SAT problems in SMT solver, it is better to model the problem with only one universal type and membership predicates for types, based on empirical results and statistical models. (Refer to "Infinity" and "GraphColoring" models in Figure 4.1 and 4.2.)
- G4: If a model has a large number of relations, it is best to use Kodkod, based on empirical evidence.

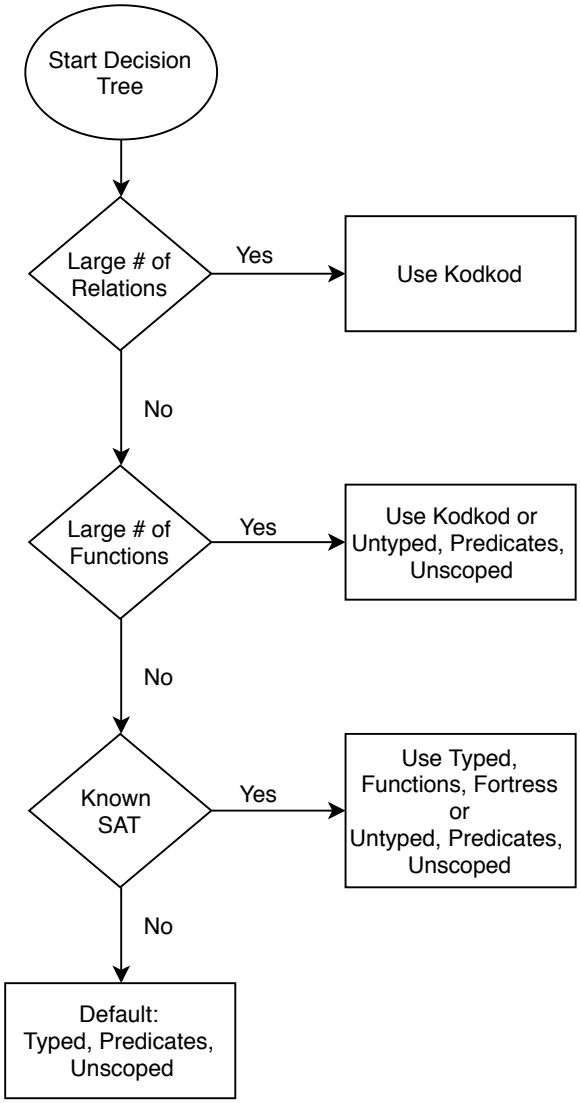

Figure 4.3: Decision tree

4.5.3 Discussion

The empirical results raise some questions regarding the internal process of SMT solvers. We had some unexpected results, such as models "Geo091" and "Geo158", which could be solved with the untyped, predicates, SMT FMF combination easily for scopes of 9 and 11, but could not be solved within the 2000 seconds time limit for a scope of 7. Also, in model "Set943", the typed, functions, the Fortress combination solved the larger scope of 11 faster than the smaller scope of 9. This test was repeated multiple times, and the same result was observed each time. These unexpected results are due to the SMT solver's behaviour.

4.6 Summary

We discussed different translation combinations and pointed out the interesting ones. We presented a set of test cases for our tool, and proposed a set of parameters that we hypothesized are effective in increasing or decreasing the performance time. Then we measured these parameters for each test case, and compared them to each other. Lastly, we presented our test results, discussed our hypotheses, and presented statistical models for the relation between parameters and the performance of the different combinations of Astra.

Chapter 5

Related Work

In this chapter, we discuss related efforts to translate Alloy to SMT-LIB. Compared to previous work, we investigate and evaluate multiple options for the translation and try to correlate them with model characteristics. While other work has evaluated the solving performance of their own translation, none of these works compare solving time for unbounded with bounded scopes in SMT solvers. Also, we start from the Kodkod interface for ease of future integration with the Alloy Analyzer. Translating from Kodkod rather than the Alloy language was easier with respect to having a more basic language to work with, but harder because we had to reverse engineer from Kodkod the types, type hierarchies, and functions of the Alloy model. We do not yet support the transitive closure operators, set cardinality or built-in types and some of these related efforts do support these operations/types.

El Ghazi *et al.* [14] describe a translation directly from Alloy to the SMT solver Yices [11]. Since Yices supports subtypes, Alloy's subtyping can be directly translated into its Yices equivalent. Partial functions in Alloy are translated to total functions in Yices by including an empty range value. The focus of their work is on using Yices' theories for Alloy's built-in types in order to leave these types unbounded. Since Yices supports λ-calculus, set operations are defined using λ-calculus. They evaluate their translation using one case study.

In El Ghazi *et al.* [12], a translation directly from Alloy to Z3 is described. It corresponds to our typed, unscoped option. They use relations at first and then do some simplifications for Alloy functions. Their work supports Alloy's built-in types and the closure operations for relations. Their results shows Z3 performed well, solving a number of problems in Alloy. Rather than using helper functions to translate the set expressions, they take a top-down approach to translation, passing arguments down to the leaf relations,

which can result in larger formulas.

Ulbrich *et al.* [23] describe a translation from Alloy to the KeY theorem prover [6] for first-order logic to check Alloy models over unbounded scopes. Their translation matches our untyped, relations, and unscoped option. They introduce helper relations to translate the set expressions using axioms similar to our constraints on the helper relations. KeY integrates automatic and interactive proof and includes support for some of Alloy's built-in types. Their translation handles transitive closure and set cardinality, but these may require interactive proof methods. Their results show that a number of Alloy assertions could be proven automatically in the KeY prover.

Reynolds et al. [20] propose a theory called Finite Cardinality Constraints (FCC) for doing finite model finding within an SMT solver. The theory is based on the EUF subset of FOL. Vakili and Day [24] report that this method did not have good performance compared to Fortress.

Bansal *et al.*[2] introduces a new theory for solving relational FOL (including set cardinality) of unbounded scope problems in SMT solvers. This is a calculus for relational logic in SMT which can be combined with their finite model finding feature. It has been implemented in CVC4 [3] and evaluated on some problems but not yet linked with Alloy for evaluation.

Alloy2B[19] is a tool that translates Alloy models to the B language [1], making a variety for B tools available for use on Alloy models including model checkers and interactive proof tools for examining a model of unbounded scope. The performance comparison is only done on one example with different scopes, and hence, does not evaluate the method's performance conclusively.

We have not yet translated the transitive closure operators. For a finite scope, it is possible to expand the meaning of transitive closure as is done by Kodkod. El Ghazi *et al.*[13] addresses this problem for unbounded scope by axiomatizing transitive closure in FOL, in an iterative manner.

Chapter 6

Conclusion

We have presented an evaluation of various options for translating relational FOL as it is represented in Kodkod to typed FOL in SMT-LIB. We considered many options for the translation including: typed vs untyped, predicates vs functions, and unbounded vs bounded scopes where the formulas are either expanded pre-solving or during SMT solving. Our results show that with the Z3 SMT solver, the typed, predicates, unscoped combination is the best combination in general for unbounded scopes; and the typed, functions, Fortress translation combination is the best for bounded scopes. We created a decision tree for choosing the best combination based on model characteristics.

We have several directions for future work. There are many interesting directions from our work to understand how model characteristics relate to solver performance, which could provide the basis for a portfolio of solvers for Alloy (perhaps based on Why3 [16]). We plan to investigate translations for the transitive closure operator, set cardinality, and the mapping of Alloy's built-in types to SMT theories. We hypothesize that the use of SMT solvers for these built-in types may provide better performance with unbounded scopes on these types. Also, for a satisfiable instance, we do not yet return the instance from the SMT solver to Alloy. This step becomes relevant when we integrate with the Alloy Analyzer, which is our next step. And we would like to broaden our analysis to include more SMT solvers or finite model finding techniques.

References

[1] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[2] Kshitij Bansal, Clark W. Barrett, Andrew Reynolds, and Cesare Tinelli. A new decision procedure for finite sets and cardinality constraints in SMT. *CoRR*, abs/1702.06259, 2017.

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer-Aided Verification (CAV)*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015.

[5] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, 2009.

[6] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, 2007.

[7] Koen Claessen and Niklas Sörensson. New techniques that improve mace-style finite model finding. In *Proceedings of the 19th International Conference on Automated Deduction: Model Computation-Principles, Algorithms, Applications*, pages 11–27. Citeseer, 2003.

[8] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. *5th International Conference on Principles of Knowledge Representation and Reasoning*, 96:148–159, 1996.

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

[10] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[11] Bruno Dutertre. Yices 2.2. In *Computer Aided Verification*, pages 737–744. Springer International Publishing, 2014.

[12] Aboubakr Achraf El Ghazi and Mana Taghdiri. Relational reasoning via SMT solving. In *Proceedings of the 17th International Conference on Formal Methods*, pages 133–148, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] Aboubakr Achraf El Ghazi, Mana Taghdiri, and Mihai Herda. First-order transitive closure axiomatization via iterative invariant injections. In *NASA Formal Methods*, pages 143–157. Springer International Publishing, 2015.

[14] Aboubakr Achraf El Ghazi and Mana Taghdiri. Analyzing alloy formulas using an SMT solver: A case study. *CoRR*, abs/1505.00672, 2015.

[15] Kuat Yessenov Greg Dennis. Forge website. http://groups.csail.mit.edu/sdg/forge. Accessed: 2018-09-30.

[16] Andrew Healy, Rosemary Monahan, and James F. Power. Predicting SMT solver performance for software verification. In *Proceedings of the Third Workshop on Formal Integrated Development Environment, F-IDE@FM*, pages 20–37, 2016.

[17] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, revised edition, 2012.

[18] Daniel Jackson and Craig A Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on software engineering*, 22(7):484–495, 1996.

[19] Sebastian Krings, Joshua Schmidt, Carola Brings, Marc Frappier, and Michael Leuschel. A translation from Alloy to B. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 71–86. Springer, 2018.

[20] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier instantiation techniques for finite model finding in SMT. In *24th International Conference on Automated Deduction*, pages 377–391. Springer, 2013.

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

[22] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *LNCS*, pages 632–647, 2007.

[23] Mattias Ulbrich, Ulrich Geilmann, Aboubakr Achraf El Ghazi, and Mana Taghdiri. A proof assistant for alloy specifications. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 422–436. Springer, 2012.

[24] Amirhossein Vakili and Nancy A. Day. Finite model finding using the logic of equality with uninterpreted functions. In *International Symposium on Formal Methods*, volume 9995 of *LNCS*, pages 677–693, 2016.