Linking Alloy with SMT-based Finite Model Finding

by

Khadija Tariq

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, 2020

Author's Declaration

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

Alloy is a well-known declarative language for modelling systems early in the development process. Currently, it uses the Kodkod library as its back-end for finite model finding (finding instances of the model by determining satisfiability for finite sets). Alloy's tool, the Alloy Analyzer, converts the Alloy model to an equivalent Kodkod problem, which is translated to boolean logic and analyzed using an off-the-shelf SAT solver by Kodkod. However, this method can often handle only problems of fairly small size sets.

We present Portus, a tool for translating Alloy into an equisatisfiable many-sorted finite model finding (MSFMF) problem of first-order logic. The MSFMF problem is evaluated by an SMT-based finite model finding method implemented in the Fortress solver, creating an alternative back-end to the Alloy Analyzer. Fortress converts the MSFMF problem into (mostly) the logic of equality with uninterpreted functions (EUF), a decidable fragment of first-order logic that is well-supported in many SMT solvers.

Portus is presented as a two-fold approach. First, we discuss the basic translation of all Alloy constructs in detail. Second, we suggest optimizations applicable to some Alloy models to improve the performance of Portus. We evaluate the effect of each optimization on Portus.

Finally, we compare the performance of Portus with Kodkod, the current solver of the Alloy Analyzer, on a corpus of Alloy models over various scopes. We classify these Alloy models based on certain characteristics and provide a hypothesis regarding the class of Alloy models Portus performs better on.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Nancy A Day, for her continuous guidance and support throughout my Master's programme. I have learned a lot from her about formal methods and academic research. I would also like to thank my readers, Joanne Atlee and Derek Rayside, for their valuable feedback.

I would also like to express my gratitude towards my colleagues without whom this journey would not have been complete. I would like to thank Ali Abbassi for his work on Astra which was the main inspiration for Portus, Dr. Amirhossein Vakili for his work on the first version of Fortress and Joseph Poremba on his work on the new version of Fortress. I would also like to thank Elias Eid for his help on profiling Alloy models to provide motivation for the optimizations in Portus and Michelle Zheng on her help in integrating Portus in the Alloy Analyzer. I would also like to thank Amin Bandali, Joesph Poremba, Elias Eid and Tamjid Hossain for their discussions on Alloy and Portus.

Finally I would like to thank my family and friends for their love and support all these years. I would not be where I am without their help and encouragement.

Table of Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Contributions . . . 3

1.2 Thesis Organization . . . 3

2 Background 4

2.1 Finite Model Finding . . . 4

2.2 Alloy and Kodkod . . . 7

2.3 Fortress . . . 8

2.4 SMT-LIB2 . . . 13

2.5 Astra . . . 13

2.6 Summary . . . 14

3 Interfacing with Fortress 15

3.1 Experimental Setup . . . 15

3.2 Fortress Model Finders . . . 16

3.3 Comparison with Kodkod . . . 22

3.4 Conclusion . . . 22

4 Basic Translation **24**

4.1 Overview . . . 24
4.2 Step 1 - Translate signatures . . . 26
4.3 Step 2 - Translate scopes . . . 29
4.4 Step 3 - Translate formulas . . . 31
4.5 Step 4 - Solve using Fortress . . . 40
4.6 Step 5 - Return instances . . . 40
4.7 Summary . . . 41

5 Optimizations **42**

5.1 Optimization of Join . . . 44
5.2 Optimization of Exact Scopes . . . 46
5.3 Optimization of Signature Hierarchy . . . 48
5.4 Optimization of Signatures . . . 50
5.5 Optimization of Relations . . . 54
5.6 Optimization of Ordering Module . . . 57
5.7 Optimization of Transitive Closure . . . 60
5.8 Optimization of Cardinality . . . 66
5.9 Optimization of Integers . . . 66
5.10 Conclusion . . . 69

6 Experimental Results **71**

6.1 Implementation . . . 71
6.2 Experimental Setup . . . 71
6.3 Performance Testing . . . 72
6.4 Correctness Testing . . . 77
6.5 Summary . . . 77

7 Related Work **79**

7.1 Astra 79

7.2 Other Alloy translations 82

7.3 Other model finding libraries 83

8 Conclusion **85**

8.1 Future Work 85

References **88**

APPENDICES **95**

A Tool versions **96**

B BNF operators **97**

C Optimizations **98**

List of Tables

3.1 TPTP problems available in Kodkod and/or included in P_{tptp} 16
3.2 Fortress model finder `v3si` with Z3 and CVC4 on P_{tptp} 17
3.3 Fortress model finders on P_{tptp} 17
3.4 Fortress model finders with simplifications on P_{tptp} 20
3.5 Effect of simplifications on Fortress model finders on P_{tptp} 21
3.6 Comparison between Fortress and Kodkod on P_{tptp} 23

5.1 Optimization of the join operator in Portus 45
5.2 Optimization of exact scopes on top-level signatures in Portus 48
5.3 Optimization of signature heirarchy in Portus 50
5.4 Optimization of signatures using constants in Portus 53
5.5 Optimization of relations using functions in Portus 56
5.6 Optimization of ordering module in Portus 60
5.7 Comparison of closure translations in Portus 65
5.8 Optimization of cardinality in Portus 68
5.9 Optimization of integers in Portus 69

6.1 Comparison between Portus and Kodkod on P_{alloyr} 74
6.2 Comparison between Portus and Kodkod on P_{alloyi} 75
6.3 Comparison between Portus and Kodkod on P_{alloyf} 76

C.1 Profiling characteristics for comparison of closure translations in Portus 99

List of Figures

4.1 Abstract syntax for an Alloy model 25
4.2 Abstract syntax for Alloy signatures 26
4.3 Alloy Signatures 27
4.4 Abstract syntax for Alloy commands 30
4.5 Abstract syntax for Alloy formulas 32
4.6 Translation of Logical Operators 33
4.7 Translation of Quantified Formulas 33
4.8 Translation of Quantified Expressions 34
4.9 Translation of Set Predicates 35
4.10 Translation of Set Operators 36
4.11 Translation of Relational Operators 36
4.12 Translation of Leaf Expressions 38
4.13 Translation of Integer Operators 39

5.1 Ordering Module in Alloy 57
5.2 Optimization of ordering module in Portus 58
5.3 Optimization of cardinality in Portus 67

7.1 Translation of Kodkod formulas in Astra 80

Chapter 1

Introduction

Finite model finding is becoming an important tool to enable automated analysis in formal verification of models. Finite model finding means searching for satisfying instances of a model within a finite scope, thus making it a decidable problem. The popular Alloy language and its Alloy Analyzer tool [30, 31] have shown the value that users find in writing models and quickly learning about their models via finite model finding analysis. Alloy has been used for analyzing the designs of high-level systems including file systems [32, 45], security [53, 69] and network configurations [59]. A comprehensive list of examples is provided in [27].

Alloy is a declarative modelling language where the model is described using sets, relations, and constraints over these relations. It is a simple, yet flexible language. Because it uses modelling elements that are very abstract (sets, relations, functions, predicates), it is suitable for describing models very early in the development process. Feedback at an early stage can help find bugs prior to a huge investment of time in a particular design direction. Recent examples of the value of early modelling are discussed in Zave [70] who modelled the CHORD protocol in Alloy and Newcombe et al. [44] who used TLA+ at Amazon.

Critical to the value of early modelling is the quick feedback from finite model finding. The Alloy Analyzer currently translates its problems into boolean logic satisfiability problems via its Kodkod [62] package and uses a variety of SAT solvers (e.g. MiniSat [56], SAT4J [36] etc.). While this method gives quick feedback for very small scopes, it reaches capacity limitations as scopes increase. Recently, Vakili and Day [65] showed that the finite model finding problem for first-order logic can be expressed in EUF, the logic of uninterpreted functions with equality [5], which is a decidable subset of many-sorted first-order

logic. The advantage of encoding finite model finding in EUF is that off-the-shelf SMT (satisfiable module theories) solvers [10] can be used to solve the problem. Vakili and Day showed that these solvers can often solve a finite model finding problem more quickly than a SAT solver likely because the structure of functions and predicates is maintained in EUF and exploited in an EUF solver. Vakili and Day implemented their approach in a solver called Fortress, which accepts problems in MSFOL as input and currently uses Z3 [20] as its underlying SMT solver. Fortress has been completely re-factored into a more powerful and robust tool by Poremba et al. [46] which provides support for integers and bit-vectors as built-in sorts. There are two options for integers in the new version of Fortress: modular arithmetic and unbounded. Although the modular arithmetic option is decidable, it can produce counterintuitive instances.

We present a tool for translating Alloy models into many-sorted first-order logic (MSFOL), which we call Portus. This method allows us to connect the Alloy Analyzer with Fortress. The translation is non-trivial because Alloy uses a relational logic over sets and includes second-order operators such as transitive closure and set cardinality, and it includes special modules such as the ordering module, which forces the elements of a set to be in a linear order. A key element in our approach is use of set membership predicates in addition to sorts to capture the set hierarchy. In addition, this method allows us to handle non-exact scopes and set cardinality. We have provided support for all of the Alloy constructs except the bit shifting operators and higher-order quantifications. Our solver is directly integrated into the Alloy Analyzer in a clone of the Analyzer repository[1]. Because we link to an SMT solver via Fortress, we investigated both a bit-vector interpretation for integers and leaving integers unbounded. We evaluated our new solver on a large corpus of Alloy models and compared its performance to Kodkod. We found that Portus performs better than Kodkod on models containing unbounded integers and relations with a range of multiplicity `one` (functions). Our work opens the door to gradually moving from bounded scopes to unbounded scopes within the SMT solver.

There have been a few efforts to link Alloy with SMT solvers before our work [23, 24, 27, 41]. However, most of these are with the goal of providing unbounded analysis of Alloy models, which is, in general, an undecidable problem. Astra [3] is a previous effort to translate Alloy to Fortress for finite model finding, but this work did not support the more interesting Alloy language features (transitive closure, set cardinality, etc.) and used a bottom-up approach to translation rather than top-down. It also was not fully integrated into the Alloy Analyzer so instances returned by Fortress were not mapped back to Alloy for visualization.

[1] Available at https://github.com/WatForm/portus.

1.1 Contributions

The main contributions of our work are as follows:

1. A translation from an Alloy model to a many-sorted finite model finding problem that covers all Alloy constructs (except the bit-shifting operators and higher-order quantifications) implemented in a tool called Portus;
2. A novel method of handling set cardinality, which utilizes the built-in sort for integers;
3. A method of checking the satisfiability of the problem at multiple finite scopes all within one solver problem;
4. Several optimizations to improve performance and analysis of their effect on Portus;
5. An extensive analysis of the performance of our tool against the state-of-the art in the Alloy Analyzer;
6. A method of checking the correctness of Portus and Kodkod interpretations via cross-checking;
7. Implementation of a simplification method for many-sorted first-order logic formulas in Fortress.

1.2 Thesis Organization

We start by providing a brief background about the Alloy language, the Fortress library and the Astra library in Chapter 2. In Chapter 3, we compare the different possibilities provided in Fortress and evaluate its performance against Kodkod. We introduce the basic translation of an Alloy model to Fortress in Chapter 4 and explore different optimizations and their effect on our library, Portus, in Chapter 5. Next in Chapter 6, we extensively evaluate the performance of Portus with Kodkod on Alloy models and test its correctness. Chapter 7 talks about related work. Finally, in Chapter 8, we conclude this thesis with a discussion about future work.

Chapter 2

Background

In this chapter, we introduce the terminology used in finite model finding and provide a brief background on the Alloy language, the Fortress library, the SMT-LIB2 language and the Astra library.

2.1 Finite Model Finding

First-order logic (FOL) or *predicate logic* is an extension of boolean logic. The basic building blocks of first-order logic consist of the truth values $\top$ (true), $\bot$ (false), the logical connectives $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\Rightarrow$ (implies), the equality symbol $=$, the quantifiers $\forall$ (for all), $\exists$ (there exists) plus an infinite sequence of variables $x, y, z, \ldots$ and some parantheses to make formulas more readable [11]. The quantifiers $\forall$ and $\exists$ range over elements of the domain M of discourse.

For the domain M of discourse, a first-order *language*[1] is defined as:

(i) a finite set of n-ary function symbols of the form $f : M^n \to M$,

(ii) a finite set of n-ary predicate symbols of the form $P : M^n \to Bool$, and

(iii) a finite set of constant symbols of the form $c \in M$.

[1]The literature of FOL typically uses the term *signature* instead of language but we avoid that due to the conflicting meaning of the term signature in Alloy.

An *interpretation* is an assignment of meaning to the symbols of a first-order language.

The *terms* of first-order logic are the smallest set of expressions containing the variables $x, y, z, \ldots$, all constant symbols and closed under the formation rule: if $t_1, \ldots, t_n$ are terms and f is an n-ary function symbol, then the expression $f(t_1, \ldots, t_n)$ is a term. An *atomic formula* is either the expression $t_1 = t_2$, where t_1 and t_2 are terms, or the expression $P(t_1, \ldots, t_n)$ where P is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms. The first-order *formulas* form the smallest set of expressions containing the atomic formulas and closed under the following formation rules:

(i) If φ, ψ are formulas, so are the expressions

$$\neg\, \varphi, \quad (\varphi \wedge \psi), \quad (\varphi \vee \psi), \quad (\varphi \Rightarrow \psi);$$

(ii) If φ is a formula and v is a variable, then $(\forall\ v : M \bullet \varphi)$ and $(\exists\ v : M \bullet \varphi)$ are formulas.

The problem of *finite model finding* means finding a satisfying interpretation for a first-order language, a finite set of first-order formulas and a finite set of elements in the domain M of discourse.

Many-sorted first-order logic (MSFOL) partitions the domain M of discourse into disjoint subsets, one for every *sort*. A language in MSFOL is defined as:

(i) a finite set of symbols called sorts,

(ii) a finite set of n-ary function symbols of the form $f : A_1 \times \cdots \times A_n \to B$,

(iii) a finite set of n-ary predicate symbols of the form $P : A_1 \times \cdots \times A_n \to \mathit{Bool}$, and

(iv) a finite set of constant symbols of the form $c \in A$

where $A, A_1, \ldots, A_n, B$ are all sorts in the language. The formulas in MSFOL must be well-sorted, which means that function and predicate symbols can only be applied to terms of the corresponding sorts and equalities can only be between terms of the same sort. The quantifiers $\forall$ and $\exists$ range over elements of sorts in the language. The formulas in MSFOL include $(\forall\ v : A \bullet \varphi)$ and $(\exists\ v : A \bullet \varphi)$ if φ is an MSFOL formula, v is a variable and A is a sort in the language.

A *domain assignment* for a language is a function mapping each sort to a non-empty, finite set of *domain elements*. The *scope* of a sort is the size of its domain. By convention,

we refer to the elements of a sort by an integer and the name of that sort. For a sort A with scope k, the domain elements can be defined as $1_A, 2_A, \ldots, k_A$.

A *many-sorted finite model finding* (MSFMF) problem means finding a satisfying interpretation for an MSFOL language, a finite set of MSFOL formulas and a domain assignment. We include integers as the only built-in sort in our MSFMF problem. Integers are assigned a scope in terms of bitwidth. For instance, a scope of 6 for integers maps all integer values from -32 to $+31$ to the integer sort in the domain assignment.

Numerous efforts have been made to convert MSFMF problems to SAT [16, 40, 49, 60, 62] and SMT [41, 50, 57, 65] problems and solve them using their respective solvers to determine if there is an interpretation that satisfies all the formulas. A *boolean satisfiability problem* (SAT) is a problem defined using propositional logic. On the other hand, *satisfiability modulo theories* (SMT) formulas provide a much richer modelling language than SAT by defining a problem with a combination of theories in first-order logic with equality [10].

Here is some terminology used in later sections:

- The logic of *equality with uninterpreted functions* (EUF) is a subset of FOL with equality and without quantifiers and variables over uninterpreted sorts [5].
- The logic of *fixed-size bit-vectors with uninterpreted functions* (UFBV) represents the theory with the standard bit-vector operators $<$, $>$, $=$, $+$, $\ldots$, quantifiers over variables of bit-vector sort and uninterpreted functions over bit-vectors [33]. A *bit-vector* is a vector of bits of a given size [35]. By convention, we express all bit-vector operators by including a subscript of the size of the bit-vector. For instance, for a bit-vector of length n, the bit-vector operators can be represented as $<_{[n]}$, $+_{[n]}$ and so on. Arithmetic for bit-vectors, also known as *modular* arithmetic, results in a wraparound effect where the bits that overflow are truncated.
- A formula is in *negation normal form* if the negation operator is applied only to atomic formulas and the only other logical operators it contains are conjunctions and disjunctions. Every formula can be converted into an equivalent formula in negation normal form.
- *Skolemization* is the process of replacing each existentially quantified variable y with $f(x_1, \ldots, x_n)$ where f is a new function symbol introduced and $x_1, \ldots, x_n$ are all the universally quantified variables preceding y in the formula. The resulting formula is not necessarily equivalent to the original one, but is *equisatisfiable* with it: it is satisfiable if and only if the original formula is satisfiable.

2.2 Alloy and Kodkod

Alloy [30, 31] is a flexible declarative language, based on first-order logic combined with relational logic, to model aspects of systems very early in the development process. It represents models using sets and relations, and constraints on these relations are expressed using relational calculus and navigation expressions (implemented via a join operator). The simplicity behind the Alloy language is that it does not distinguish between sets and scalars and treats all of them as relations. Sets are expressed as unary relations and scalars as singleton sets. This uniformity makes it easier to write constraints by using the same operator for scalars, sets and relations.

Basic sets are declared using *signatures*. We use the terms set and signature interchangeably. A set hierarchy can be created using the `extends` keyword in Alloy. Subsets can be declared to be mutually disjoint or not. Relations are declared as *fields* of signatures as in:

```
1  sig A {
2      f1: B,
3      f2: B -> C
4  }
```

which means there are two relations: `f1: A -> B` and `f2: A -> B -> C`. We call `A -> B` a *compound type expression* even though Alloy is largely untyped. `A -> B` means `A` is the domain of the relation and `B` is the range of the relation. Relations are flat, meaning `f2` is a ternary relation. More details about the Alloy modelling language are provided in later chapters as needed.

An *instance* (or *counterexample*) comprises of the elements in sets associated with the signatures and the tuples in relations associated with the fields. An instance also might contain an interpretation for any skolem constants or functions produced during skolemization. The Alloy Analyzer can be used either to find sample instances of a model using the `run` command or to check if the model violates a given property and produce a counterexample using the `check` command. This analysis is fully automated but it only checks for solutions consisting of sets of finite scope.

Kodkod [61, 62] is a library for finite model finding for a more basic relational logic than Alloy. Given an Alloy model, the Alloy Analyzer converts it to an untyped Kodkod problem consisting of a universe made up of all the elements, a set of relation declarations, each with a lower and upper bound on its tuples, and a formula. A technique, called *atomization*, is used to refactor the set heirarchy into a flat collection of disjoint atomic types [22]. Relations and constraints are refactored and broken down to use these types

appropriately. Compared to lifting subtypes to supertypes, which would result in inflation of relation size, this approach results in minimizing the size of relations and formulas.

An n-ary relation in Kodkod is encoded using an n-dimensional boolean matrix. Kodkod translates formulas to propositional logic by applying operators on these matrices, skolemizing the existential quantifiers and expanding the universal quantifiers before feeding the final constraint over boolean constants to an off-the-shelf SAT solver for evaluation. Second-order operators such as transitive closure and cardinality are expressed by fully expanding their meaning for the finite scope. Kodkod provides support for many SAT solvers including MiniSAT [56], SAT4J [36], Glucose [7] and (P)Lingeling [13]. Particular effort has been made in Kodkod to do symmetry reductions [61]. Kodkod (although not the Alloy language) provides special support for partial instances, which are instances where part of the solution is already known. A partial instance can be specified by setting the lower bound of a relation to include only the tuples in the partial instance.

2.3 Fortress

Fortress is a library that takes an MSFMF problem and checks its satisfiability by invoking an SMT solver. The first version of Fortress, presented as Vakili and Day's work in [65], has been completely re-factored to a more powerful and robust tool by Poremba et al. [46]. Unlike the first version of Fortress, the new version includes domain elements as terms and provides support for the built-in sorts integers and bit-vectors. Domain elements can be used in MSFOL formulas to specify a partial instance.

The MSFMF problem provided as input to Fortress may include the built-in sort for integers. Fortress provides two options for handling integers: modular arithmetic and unbounded. For the modular arithmetic option, Fortress converts an MSFMF problem containing integers to a combination of the logic of EUF and UFBV, which we denote as *EUFBV*. For the unbounded option, Fortress does not use the scope for integers and converts an MSFMF problem to an MSFOL problem because it may contain quantification over unbounded integers. An MSFMF problem not containing the built-in sort for integers or bit-vectors is translated to EUF by Fortress regardless of which option for integers is chosen.

Fortress translates an MSFMF problem to the logic of EUFBV through a series of transformations. We use a small example to illustrate the five steps performed by Fortress during this translation. Consider the following set of formulas:

1. $\exists x : A \bullet \neg \ (\ R(x) \vee \exists y : A \bullet f(x) \neq f(y)\ )$
2. $\forall x : Int \bullet g(x) > 3$

where $R : A \rightarrow Bool$ is a predicate symbol, $f : A \rightarrow A$ and $g : Int \rightarrow Int$ are function symbols, the scope of sort A is 3 and the scope (or bit-width) of the built-in integer sort Int is 3.

Step 1 - Finitize Integers. For the modular arithmetic option, Fortress replaces all occurrences of the built-in integer sort Int with the built-in bit-vector sort $BitVec$. All integer operators are replaced with their corresponding bit-vector operators. Applying these changes to the formulas above results in:

1. $\exists x : A \bullet \neg \ (\ R(x) \vee \exists y : A \bullet f(x) \neq f(y)\ )$
2. $\forall x : BitVec_{[3]} \bullet g'(x) >_{[3]} 3_{[3]}$

where $g' : BitVec_{[3]} \rightarrow BitVec_{[3]}$ is the function symbol used to replace g in the MSFMF problem. If the option for unbounded integers is chosen or if there are no integers in the MSFMF problem, this step is skipped. The complexity of this step is linear with respect to the size of the FOL formulas.

Step 2 - Normalize. Each formula is converted to negation normal form and then skolemized to remove existential quantifiers. Applying these transformations to the formulas in Step 1 results in:

1. $\neg R(sk) \wedge \forall y : A \bullet f(sk) = f(y)$
2. $\forall x : BitVec_{[3]} \bullet g'(x) >_{[3]} 3_{[3]}$

where $sk \in A$ is a constant symbol introduced as the result of skolemization. The complexity of this step is linear with respect to the size of the FOL formulas.

Step 3 - Ground Formulas. Each universally quantified formula is instantiated with the domain elements corresponding to the sort of each quantified variable. The variables in quantified formulas ranging over the built-in sorts are not expanded regardless of which option for integers is chosen. Given the domain elements of A as 1_A, 2_A and 3_A, grounding the formulas from Step 2 results in:

1. $\neg R(sk) \wedge f(sk) = f(1_A) \wedge f(sk) = f(2_A) \wedge f(sk) = f(3_A)$
2. $\forall x : BitVec_{[3]} \bullet g'(x) >_{[3]} 3_{[3]}$

The complexity of this step is exponential with respect to the number of nested universal quantifiers.

Step 4 - Add Range Formulas. Range formulas are added for all constant and function symbols constraining their values to the domain elements of their respective sorts. No range formulas are added for the symbols containing the built-in sorts for integers and bit-vectors. This step is the key contribution of Fortress. We add the constraints:

$$
\begin{gathered}
sk = 1_A \vee sk = 2_A \vee sk = 3_A \\
f(1_A) = 1_A \vee f(1_A) = 2_A \vee f(1_A) = 3_A \\
f(2_A) = 1_A \vee f(2_A) = 2_A \vee f(2_A) = 3_A \\
f(3_A) = 1_A \vee f(3_A) = 2_A \vee f(3_A) = 3_A
\end{gathered}
$$

The complexity of adding range formulas is exponential with respect to the arity of the function symbols.

Step 5 - Add Domain Element Formulas. The last step asserts that the domain elements are mutually distinct[2].

The logic of EUF is a decidable fragment of first-order logic and its complexity is NP-complete [5, 35]. The theory of UFBV with quantification over bit-vector sort is decidable and its complexity is NEXPTIME-complete [67, 68]. EUFBV is not known to be decidable because it may include uninterpreted functions that map bit-vectors to uninterpreted sorts or vice versa. However, since the original problem provided as input to Fortress is an MSFMF problem, the problem that results from Fortress with the bit-vector interpretation for integers is decidable i.e. there is a finite set of possible satisfying interpretations for an MSFMF problem. For the unbounded integers option in Fortress, the resulting MSFOL problem is not necessarily decidable.

In the new version of Fortress, each action in the translation to EUFBV as mentioned above is represented as an independent transformer, which grants the user freedom to modify or change the order of these transformations. Fortress has also added a new feature of creating an interactive process to communicate with the SMT solver. It provides support for two SMT solvers, Z3 and CVC4, and support for any other solver can be added with very little effort. If the MSFMF problem is satisfiable, Fortress can retrieve an interpretation for the constants, function and predicate symbols, find another interpretation and count the number of interpretations of an MSFMF problem.

The new Fortress applies new symmetry breaking schemes and sort inference in order to increase the efficiency of the solver. Model finders exploit symmetries in the problem by adding *symmetry breaking formulas* [18], that disallow redundant interpretations of the problem as solutions to reduce the search space. Symmetry breaking has been used by

[2] In SMT-LIB, this constraint is written simply as (`distinct` 1_A 2_A 3_A).

numerous model finders [16, 49, 55, 62]. *Sort inference* is the process of finding a more generally sorted problem. It is well known that sort information helps improve the efficiency of the solver by allowing symmetry breaking formulas to be stronger. It is shown to be beneficial by Claessen and Sörensson [16] and used by Reger et al. [49] in the Vampire theorem prover.

In the context of finite model finding, Claessen and Sörensson first introduced a symmetry breaking technique by assuming an ordering on the domain elements [16]. This technique applies symmetry reduction on constants first and subsequent constant and function symbols have gradually more freedom in their possible values. The first and the new version of Fortress both use this technique to optimize the range formulas generated in Step 4. The constraints in the example mentioned above can be reduced to:

$$\begin{gathered} sk = 1_A \\ f(1_A) = 1_A \vee f(1_A) = 2_A \\ f(2_A) = 1_A \vee f(2_A) = 2_A \vee f(2_A) = 3_A \\ f(3_A) = 1_A \vee f(3_A) = 2_A \vee f(3_A) = 3_A \end{gathered}$$

However, Claessen and Sörensson's symmetry breaking technique [16] considers sorts in isolation and is applicable for constants and mono-sorted functions only. A *mono-sorted* function has all input sorts equal to the result sort and is of the form $f : A \times \cdots \times A \to A$. In the new Fortress, Poremba et al. [46] introduced symmetry breaking schemes for functions by classifying them into two categories – range-domain independent and range-domain dependent. A function $f : A_1 \times \cdots \times A_n \to B$ is *range-domain independent* if the result sort B is distinct from each of its input sorts $A_1, \ldots, A_n$. Otherwise, the function is *range-domain dependent*. A *ladder symmetry breaking scheme* is also introduced for predicates.

Consider sorts A and B with scopes of 3 each. For a range-domain independent function $f : A \to B$, $f(1_A), \ldots, f(3_A)$ can be considered as constants from the perspective of B since the range and domain of the function are independent from each other resulting in the following symmetry breaking formulas:

$$\begin{gathered} f(1_A) = 1_B \\ f(2_A) = 1_B \vee f(2_A) = 2_B \\ f(3_A) = 1_B \vee f(3_A) = 2_B \vee f(3_A) = 3_B \end{gathered}$$

For a range-domain dependent function $f : A \times B \to A$, the possible values in the range of the function are dependent on the values used in the domain of the function. A value for B is held constant resulting in the following symmetry breaking formulas:

$$f(1_A, 1_B) = 1_A \vee f(1_A, 1_B) = 2_A$$
$$f(2_A, 1_B) = 1_A \vee f(2_A, 1_B) = 2_A \vee f(2_A, 1_B) = 3_A$$
$$f(3_A, 1_B) = 1_A \vee f(3_A, 1_B) = 2_A \vee f(3_A, 1_B) = 3_A$$

For a predicate $P : A \rightarrow Bool$, the ladder scheme results in the following symmetry breaking formulas:

$$P(2_A) \Rightarrow P(1_A)$$
$$P(3_A) \Rightarrow P(2_A)$$

because a predicate represents a set and all sets of the same size can be considered equivalent.

The new Fortress provides a choice of six model finders differing from each other in their symmetry breaking schemes and the order in which symmetry breaking is applied:

- `v0` contains no symmetry breaking optimizations;
- `v1` applies symmetry breaking on constants and then mono-sorted functions same as the Claessen and Sörensson's symmetry breaking technique [16] discussed above;
- `v2` applies symmetry breaking on constants, then on all functions and then on all predicates;
- `v2si` is similar to `v2` with sort inference;
- `v3` applies symmetry breaking on constants, then on all mono-sorted functions, then on all other functions and then on all predicates;
- `v3si` is similar to `v3` with sort inference;

The previous Fortress shows promising results when comparing against Kodkod [65]. The new version of Fortress has been more thoroughly tested both for its performance and correctness. The next chapter compares the performance of new Fortress with Kodkod. Moreover, it also compares the performance of all the model finders and SMT solvers provided by the new Fortress library. From now onwards, any mention of Fortress in the thesis refers to the new version of Fortress.

2.4 SMT-LIB2

Fortress translates the MSFMF problem to SMT-LIB2 – the SMT-LIB standard, version 2.0 [48]. Example of theories supported by the SMT language include the theory of real numbers, integers and various data structures like arrays, lists, bit-vectors and so on. The SMT solvers supported by the Fortress library are Z3 and CVC4.

Z3 is an efficient SMT solver, with an integrated CDCL-based SAT solver, developed by Microsoft Research [20]. CVC4 is an automated theorem prover for SMT problems developed by NYU and U Iowa [9]. Both solvers provide support for rational and integer linear arithmetic, fixed-size bit-vectors, uninterpreted functions and quantifiers. Sorts defined in Z3 and CVC4 are mutually disjoint and there is no subtyping.

2.5 Astra

A previous effort to translate Alloy to SMT called Astra [3] translates a Kodkod (not Alloy) problem to an MSFMF problem with the previous version of Fortress as its solver. It traverses the Kodkod formula in a bottom-up manner. Since Kodkod is untyped and atomized to the signatures at the lowest level of the signature heirarchy, Astra has to reverse-engineer the sets, set hierarchies and relations from the provided formulas.

Astra does not provide support for:

1. the binary relational operators: override (`++`) and product (`->`);
2. the unary relational operators: transpose (`~`), transitive closure (`^`) and reflexive transitive closure (`*`);
3. the integer comparison operators (`=`, `>=`, `<=`, `>`, `<`);
4. the integer arithmetic operators (`+`, `-`, `*`, `/`, `%`);
5. the cardinality operator (`#`);
6. multiplicity formulas: `some e`, `no e`, `lone e` and `one e` where `e` can be any expression representing a set or relation;
7. negated quantifier formulas (explained in Section 7.1.1);
8. the use of the ordering module;

9. non-exact scopes;

10. retrieving an instance in the case of satisfiability.

Astra shows promising results when comparing against Kodkod [3] on TPTP benchmarks. However, it lacks extensive testing for performance against Kodkod on Alloy benchmarks and for its correctness. Our approach is significantly different from Astra and has been extensively tested for its performance and correctness. Our library, Portus, also provides support for each missing functionality listed above. Moreover, Portus is fully integrated with the Alloy GUI including the option to visualize the returned instance and get the next instance.

2.6 Summary

An MSFMF problem means finding a satisfying interpretation for a language, a finite set of MSFOL formulas and a domain assignment. Alloy is a language used to verify properties of a system. The Alloy Analyzer uses the Kodkod library as its back-end, which uses a SAT solver to analyze a given problem. Fortress is a finite model finder that takes an MSFMF problem, translates it to (mostly) EUF and solves using an SMT solver. Since previous results show that Fortress is comparable to Kodkod, we propose that Fortress be used as an alternative back-end to Alloy. A previous effort to translate Alloy models to Fortress called Astra has missing functionality and has not been thoroughly tested.

Chapter 3

Interfacing with Fortress

This chapter explores the different options available in Fortress in order to choose the settings for the evaluation of our library, Portus, in later chapters. We compare the performance of Fortress to Kodkod on selected benchmarks similar to what was done for the previous version of Fortress [65].

3.1 Experimental Setup

The problems used to compare the performance of Fortress model finders to Kodkod are taken from the TPTP library [58] for automated theorem provers. TPTP problems are categorized into seven main fields with a total of forty-six domains. Although TPTP benchmarks are not a priority in our case, it is the only input format readily available to be tested on both Fortress and Kodkod. Fortress can accept TPTP as input. Torlak and Jackson manually translated some TPTP problems to Kodkod in [61] – these mostly come from a single field.

We choose a small representative subset of TPTP problems, which we call P_{tptp}[1], spanning multiple fields. There is at most one problem per domain and we evaluate each problem on increasing scopes. All problems are in unsorted first-order logic and are unsatisfiable. The TPTP problems in our set P_{tptp} are shown in Table 3.1. As shown in the table, some of the problems from Torlak et al. have been excluded from our analysis because of our limit of only one problem per domain. We are limited in our comparison by which problems are available in the Kodkod format.

[1] Available at https://github.com/WatForm/portus-tests.

	Field / Domain	Available in Kodkod	Included in P_{tptp}
alg195	Maths / Gen Algebra	✓	
alg197	Maths / Gen Algebra	✓	
alg212	Maths / Gen Algebra	✓	✓
com008	CS / Computing Theory	✓	✓
geo091	Maths / Geometry	✓	✓
geo092	Maths / Geometry	✓	
geo115	Maths / Geometry	✓	
geo158	Maths / Geometry	✓	
med007	Science & Eng. / Medicine	✓	
med009	Science & Eng. / Medicine	✓	✓
mgt036	Social Sciences / Management		✓
num374	Maths / Number Theory	✓	✓
num378	Maths / Number Theory	✓	

Table 3.1: TPTP problems available in Kodkod and/or included in P_{tptp}

All experiments were run on Intel®Xeon®CPU E3-1240 v5 @ 3.50 GHz with Ubuntu 16.04 64-bit with up to 64GB of user memory. The scope is chosen such that at least one tool takes more than 30 seconds. We repeat each experiment five times and report the average with a time limit of 30 minutes on each process. Each timeout is assigned a value of 1800 during the total time calculation for each tool. Information about the version of each tool we used is available in Appendix A.

3.2 Fortress Model Finders

Fortress provides users with a choice of six model finders and two SMT solvers as discussed in Section 2.3. This section compares these possibilities on the problems in P_{tptp} in order to choose a model finder and an SMT solver to continue with our evaluation.

Table 3.2 compares the results of the `v3si` model finder with Z3 and CVC4 on the problems in P_{tptp}. The results demonstrate that Z3 performs significantly better on all benchmarks. As a result, we choose Z3 as the SMT solver in Fortress for the rest of our experiments and our evaluation in later chapters.

Using Z3 as the SMT solver, the time taken (in seconds) by each Fortress model finder

	alg212	com008	geo091	med009	mgt036	num374
Scope	8	12	6	19	13	6
Z3 (s)	2.39	37.06	4.82	20.28	6.32	7.25
CVC4 (s)	1391.83	t/o	t/o	t/o	23.85	t/o

Table 3.2: Fortress model finder `v3si` with Z3 and CVC4 on P_{tptp}

	Scope	v0	v1	v2	v2si	v3	v3si
alg212	8	t/o	4.43	4.39	4.42	4.42	2.39
	9	t/o	27.96	27.94	28.11	27.96	28.05
	10	t/o	184.84	183.98	183.27	185.19	183.32
com008	12	t/o	37.34	37.16	37.17	37.08	37.06
	13	t/o	41.40	41.24	41.29	41.35	41.51
	14	t/o	67.24	66.89	66.77	67.34	66.75
geo091	6	32.79	15.27	15.17	3.82	15.15	4.82
	7	157.83	197.59	197.83	47.91	198.12	35.82
	8	t/o	780.69	785.81	224.26	786.48	138.43
med009	19	84.21	20.58	20.18	20.23	20.24	20.28
	21	108.07	38.49	37.71	37.57	37.53	37.69
	23	234.18	45.47	44.91	44.75	44.78	44.58
mgt036	13	37.65	6.58	6.50	8.91	6.59	6.32
	14	71.86	32.92	32.99	20.55	33.05	34.26
	15	83.63	40.04	39.72	38.85	39.77	27.34
num374	5	48.82	0.91	0.90	0.90	0.91	0.90
	6	t/o	7.23	7.26	7.22	7.25	7.25
	7	t/o	40.70	40.26	40.55	40.55	40.58
Best out of 18		0	0	5	5	1	9
Total Time (s)		17 059.04	1589.68	1590.84	856.55	1593.76	757.35

Table 3.3: Time taken (in seconds) by Fortress model finders on P_{tptp}
(The highlighted cell indicates the best result in each row. t/o = timed out after 1800s.)

to solve our set of TPTP benchmarks, P_{tptp}, is shown in Table 3.3 where the highlighted cells indicate the lowest time for each benchmark. The results demonstrate that the `v0` Fortress model finder performs the worst on all benchmarks with a total of 9 timeouts. This result is to be expected since `v0` does not have any symmetry breaking optimizations. The other three model finders, `v1`, `v2` and `v3` have similar performance to each other on all benchmarks. However, the sort inference model finders, `v2si` and `v3si` have slightly better performance on all benchmarks and the best overall performance compared to the other model finders. One notable result is the performance of the sort inference model finders on `geo091`, where both `v2si` and `v3si` are able to infer one extra sort in the given problem. No extra sorts are inferred by `v2si` or `v3si` on the other benchmarks.

3.2.1 Contributions to Fortress

We investigate the effect of some additional simplifications on formulas after grounding within Fortress, similar to what has been done in [65]. Fortress simplifies logical connectives if the truth value of any of its operands is known e.g. $\top \vee t$ is simplified to $\top$ and $\bot \vee t$ to t. In addition to that, we introduce two key simplifications:

Learned literals simplification: A *literal* is defined as an atomic formula or its negation:

$$t_1 = t_2, \quad \neg\,(t_1 = t_2), \quad R(t_1, \ldots, t_n), \quad \neg R(t_1, \ldots, t_n)$$

where $t_1, t_2, \ldots, t_n$ are terms and R is an n-ary predicate symbol. If a formula in an MSFMF problem is a literal, its truth value can be learned or discovered, i.e. atomic formulas are learned as $\top$ and their negations as $\bot$. A syntactic ordering is applied on formulas so $t_1 = t_2$ is equivalent to $t_2 = t_1$. All instances of a learned literal can be replaced by its learned truth value in all formulas of the MSFMF problem.

Domain elements simplification: As mentioned in Section 2.3, all domain elements are mutually distinct. Equality comparison between domain elements can be simplified e.g. $1_A = 1_A$ can be simplified to $\top$ and $1_A = 2_A$ to $\bot$.

Conjunctions in the MSFMF problem are split into separate formulas recursively prior to simplification in order to maximize the number of literals that can be discovered. Simplification is done in multiple passes until no new literals are discovered.

We illustrate these simplifications using a simple example. Consider the following set of formulas after grounding:

1. $\neg\ R(x) \vee (x = 1_A \vee x = 2_A)$
2. $R(x) \vee 1_A = 2_A$

where $R : A \to Bool$ is a predicate symbol, 1_A and 2_A are the domain elements of sort A, and $x : A$ is a constant symbol. We start with an empty set of learned literals. In the first pass, there is no possible simplification to be done on 1 and no literal can be learned. $1_A = 2_A$ is simplified to $\bot$ in the formula on 2 using the domain elements simplification and further simplified to $R(x)$ using the simplifications over logical connectives. $R(x)$ is then added to our set of learned literals. In the second pass, formula 1 is simplified to $\neg\ \top \vee (x = 1_A \vee x = 2_A)$ using the learned literals simplification and further simplified to $x = 1_A \vee x = 2_A$ using the simplifications over logical connectives. No other simplification is done and no new literals are discovered in the third pass. The result is the following set of formulas after simplification:

1. $x = 1_A \vee x = 2_A$
2. $R(x)$

The time taken (in seconds) by each model finder to solve our set of TPTP benchmarks, P_{tptp}, after including these simplifications is shown in Table 3.4 where the highlighted cells indicate the lowest time for each benchmark. Table 3.5 shows the results representing the percentage change in performance after simplifications as compared to the results before simplifications in Table 3.3. An improvement in performance resulting in lower time and hence, a value less than 100, is indicated by a blue entry. The `v3si` column for `com008` and scope 13, for example, shows that the total time after simplifications is 85% of the time before simplifications. The red entries indicate when the model finders take more time with simplification than without.

The results demonstrate that the effect of simplifications on the performance of model finders seems to be dependent on the problem. For some of the benchmarks like `com008` and `geo091`, the additional simplifications clearly produce a significant positive effect whereas, for `alg212`, none of the scopes or model finders show a positive effect. One notable result is the performance of all model finders on `geo091` for scope 8, especially `v0` which timed out before simplifications. Given the results, we choose the `v2si` model finder with simplifications for our evaluation against Kodkod. From now onwards in the thesis, we use the versions of Fortress model finders to refer to the versions with simplifications.

	Scope	v0	v1	v2	v2si	v3	v3si
alg212	8	t/o	5.01	5.02	5.03	5.00	5.01
	9	t/o	31.61	31.80	31.51	31.60	31.59
	10	t/o	198.94	197.48	198.07	198.59	197.62
com008	12	t/o	32.17	32.08	32.12	32.37	32.20
	13	t/o	35.06	34.81	35.10	39.45	35.03
	14	t/o	66.52	66.58	66.76	67.78	66.72
geo091	6	51.53	14.58	14.49	3.1	14.52	2.91
	7	136.43	161.79	162.13	26.96	162.38	24.85
	8	624.00	374.69	380.81	73.26	383.48	334.13
med009	19	184.11	20.85	20.73	20.58	20.57	20.65
	21	126.17	37.46	37.17	37.11	37.46	37.32
	23	311.28	50.34	50.38	49.84	50.91	50.16
mgt036	13	16.99	22.87	22.81	11.14	22.88	15.57
	14	30.03	18.9	18.98	14.74	18.92	14.97
	15	43.17	27.00	27.24	20.75	27.07	21.07
num374	5	9.10	0.84	0.83	0.84	0.84	0.84
	6	t/o	6.81	6.82	6.83	6.81	6.82
	7	t/o	43.67	43.70	43.90	43.82	43.90
Best out of 18		0	3	4	7	3	2
Total Time (s)		15 932.81	1149.11	1153.86	677.64	1163.45	941.36

Table 3.4: Time taken (in seconds) by Fortress model finders with simplifications on P_{tptp} (The highlighted cell indicates the best result in each row. t/o = timed out after 1800s.)

	Scope	v0	v1	v2	v2si	v3	v3si
alg212	8	t/o	113	114	114	113	210
	9	t/o	113	114	112	113	113
	10	t/o	108	107	108	107	108
com008	12	t/o	86	86	86	87	87
	13	t/o	85	84	85	95	84
	14	t/o	99	99	99	99	99
geo091	6	157	95	96	81	96	60
	7	86	82	82	56	82	69
	8	35	48	48	33	49	241
med009	19	219	101	103	102	102	102
	21	117	97	98	98	99	99
	23	133	111	112	111	114	112
mgt036	13	45	358	351	125	347	246
	14	42	57	58	72	57	44
	15	52	67	69	53	68	77
num374	5	19	92	92	93	92	93
	6	t/o	94	94	95	94	94
	7	t/o	107	109	108	108	108
Total Difference (%)		93	72	73	79	73	124

Table 3.5: Percentage effect of simplifications on Fortress model finders on P_{tptp} (The blue entries indicate an improvement in performance compared to Table 3.3 by percentage. The red entries indicate percentage decrease in performance. t/o = timed out after 1800s.)

3.3 Comparison with Kodkod

Table 3.6 presents the comparison results for Kodkod, Fortress and Alloy on the problems in P_{tptp} where the highlighted cells indicate the lowest time for each benchmark. The `Fortress` column represents the results of the `v2si` model finder (with simplifications). The `Kodkod` column represents the results for the hand-crafted problems included in the Kodkod library. These problems are manually translated by Torlak and have been optimized to work best with Kodkod's infrastructure whereas, Fortress uses the problems without any modifications. As an alternative to the customized Kodkod problems, we used a translator from the previous Fortress library to convert the problems in P_{tptp} to Alloy. The Alloy GUI is then used to convert these problems to Kodkod for each scope and the results are provided in the `Alloy` column. The time taken to convert a TPTP problem to Kodkod is not included in the calculation. In order to ensure the diversity of problems in P_{tptp}, we have included an extra problem from a new field and a new domain. Since it is not available in the Kodkod format, we have included its results separately for the `Fortress` and `Alloy` columns only.

The results indicate that Fortress performs best overall, with the lowest time on 9 out of 18 benchmarks with only one timeout. Kodkod performs best on 7 benchmarks with two timeouts and 3 benchmarks are not available to be tested because they are not included in the set of manually translated TPTP problems in the Kodkod library. Alloy performs best on only 2 of the benchmarks with a total of 4 timeouts. Another observation from our results is the significant gap in performance between the hand-crafted Kodkod problems and the Kodkod problems generated directly from the TPTP problems via the Alloy GUI. Taking a closer look at the customized Kodkod problems shows that the formulas have been simplified and the context of the problem used to specify partial instances for each problem, giving Kodkod unfair advantage over the other tools. In the comparison between the `Fortress` and `Alloy` columns, Fortress performs better than the direct uncustomized translation to Kodkod on 12 out of 18 benchmarks.

3.4 Conclusion

Based on our evaluation, we choose to continue with the sort inference model finder `v2si` with simplifications and the Z3 SMT solver. Comparing the performance of Fortress to Kodkod gives promising results. However, more extensive testing is needed, particularly on Alloy models, in order to gauge if Fortress can be an alternative back-end to Alloy.

	Scope	Fortress (v2si)	Kodkod	Alloy
alg212	8	5.03	172.45	670.09
	9	31.51	1482.91	t/o
	10	198.07	t/o	t/o
com008	12	32.12	2.01	17.73
	13	35.10	2.75	52.67
	14	66.76	2.84	113.16
geo091	7	26.96	1.14	22.92
	8	73.26	2.60	121.23
	9	t/o	16.31	90.87
med009	19	20.58	0.37	0.39
	21	37.11	0.70	0.54
	23	49.84	1.07	0.84
num374	5	0.84	11.65	96.47
	6	6.83	206.53	t/o
	7	43.9	t/o	t/o
No. of timeouts		1	2	4
Best out of 15		6	7	2
Total Time (s)		2427.91	5503.33	8386.91
		×1	×2.26	×3.45

mgt036	13	11.14	n/a	29.68
	14	14.74	n/a	49.27
	15	20.75	n/a	76.98
Best out of 18		9	n/a	2
Total Time (s)		2474.54	n/a	8542.84

Table 3.6: Time taken (in seconds) by Fortress and Kodkod on P_{tptp}
(The highlighted cell indicates the best result in each row. t/o = timed out after 1800s. The upper table contains the results for the problems provided with the Kodkod library. The lower table contains the results for the problems not available in the Kodkod format.)

Chapter 4

Basic Translation

This chapter outlines the approach taken by our library, Portus[1], to translate an Alloy model to an MSFMF problem, solve it using an appropriate solver and produce an instance or counterexample if needed. Portus uses the Fortress library to represent the MSFMF problem.

4.1 Overview

This chapter introduces the basic rules for translating an Alloy model to an MSFMF problem and provides a baseline for the optimizations presented in the next chapter. The abstract syntax for an Alloy model is included in Figure 4.1. An Alloy model consists of imported modules, signature declarations, facts (formulas or constraints on the model) and command declarations. The abstract syntax for Alloy signatures, commands and formulas is included in Figure 4.2, Figure 4.4 and Figure 4.5 respectively and discussed in further detail in the later sections.

The Alloy Analyzer provides convenient methods to parse an Alloy model in various forms, thus allowing Portus to process the input in an intermediary format. For an Alloy model, Portus extracts the signature declarations and the list of commands using the Alloy Analyzer. Each command includes an Alloy formula and a mapping from each signature to its scope. For the analysis of a command, Portus conjoins the signature facts, the facts of the model and the formula of the command. As discussed in Section 2.2,

[1]The name is inspired from the charm incantation to transform objects into Portkeys in the famous fantasy novel series Harry Potter by J. K. Rowling [52].

alloyModel ::= import* paragraph*

import ::= `open` identifier [`as` identifier]
paragraph ::= sigDecl | factDecl | cmdDecl
factDecl ::= `fact` { formula* }

Figure 4.1: Abstract syntax for an Alloy model (taken from Jackson [31])
(Operators for this BNF notation are defined in Appendix B)

the Alloy language provides a choice of two commands: `run` and `check`. For an Alloy formula `f`, the `run` command checks for the satisfiability of `f` and the `check` command checks for the satisfiability of `not f`. For simplicity, we describe the translation in later sections under the assumption of one `run` or `check` command per file.

The correctness of our method is determined by extensive testing and the details are mentioned in Chapter 6. The methodology of Portus is decomposed into five steps:

Step 1 - Translate Signatures. The first step extracts the list of Alloy signatures, a signature hierarchy and relation declarations and translates them into a language in MSFOL. A finite set of MSFOL formulas might also be produced during this stage.

Step 2 - Translate Scopes. In the second step, the scope of each signature is used to either set a scope for a sort or add an MSFOL formula.

Step 3 - Translate Formulas. The third step performs a top-down traversal of the abstract syntax tree (AST) for the Alloy formula which results in a finite set of MSFOL formulas.

Step 4 - Solve using Fortress. In the fourth step, Portus uses a Fortress model finder to convert the MSFMF problem to SMT-LIB2 and use an SMT solver to check its satisfiability.

Step 5 - Return Instances. If the problem is satisfiable, the last step takes the Fortress interpretation and converts it to an Alloy instance or counterexample.

As we present our translation, we describe the meaning of the relevant parts of the Alloy language. We define our translation operator as $[\![\cdot]\!]$, which takes in an Alloy formula and returns an MSFOL formula. In our description of the translation, we use the following auxiliary functions:

- *sort*$_i$(`e`) takes an Alloy expression `e` of compound type $T_1 \times \cdots \times T_i \times \cdots \times T_n$ and returns the MSFOL sort representing the top-level signature of type T_i. For example, for the type expression `A -> B` in Alloy, where `A` is a top-level signature and `B` is a subsignature of the top-level signature `C`, *sort*$_1$(`A -> B`) returns the MSFOL sort representing `A` and *sort*$_2$(`A -> B`) returns the MSFOL sort representing `C`.
- *arity*(`e`) returns the size of a tuple in the set or relation representing expression `e` in Alloy.
- *scope*(`s`) returns the size of signature `s` in the Alloy model.

sigDecl ::= [`abstract`] [mult] `sig` identifier,$^+$ [sigExt] { formula* }
sigExt ::= `extends` sig | `in` sig [`+` sig]*

Figure 4.2: Abstract syntax for Alloy signatures (taken from Jackson [31])
(Operators for this BNF notation are defined in Appendix B)

4.2 Step 1 - Translate signatures

The abstract syntax for signature declarations is shown in Figure 4.2 and an example of each type of signature declaration is demonstrated in Figure 4.3. A *signature* in Alloy introduces a set. For example, the signature on line 1 of Figure 4.3 introduces a set named `A`. Since `A` is declared independently of any other signature, it is called a *top-level* signature. Portus translates all top-level signatures to sorts in Fortress.

Each signature has a scope associated with it, which usually sets an upper bound on the size of that set[2]. An instance or counterexample produced can have any number of elements from zero to the scope value for that set. To handle these non-exact scopes (as well as scopes of size 0 since a sort in MSFOL must have a domain size of at least one as defined in Section 2.1), for every signature, we introduce a predicate for set membership. More details on how to handle scopes will be discussed in Section 4.3. For set `A`, we introduce a predicate in MSFOL:

$$inA : A \rightarrow Bool$$

[2]It is possible for a signature to have an exact scope.

```
1  sig A {
2      f : e
3  }
4  sig A1 in A { }
5  sig A2, A3 extends A {}
6
7  abstract sig B { }
8  sig B1 extends B { }
9  sig B2 extends B { }
```

Figure 4.3: Alloy Signatures

where A is the sort representing set `A`.

A subset signature can be declared as shown on line 4 of Figure 4.3. For each subset signature, we introduce a predicate to represent set membership for that set, similar to the one created for top-level signatures. For set `A1`, we create a predicate in MSFOL:

$$inA1 : A \rightarrow Bool$$

To relate `A` and `A1`, we introduce the formula:

$$\forall x : A \bullet inA1(x) \Rightarrow inA(x)$$

to make sure that any element included in set `A1` is also included in `A`.

A *subsignature* or *extension* of a set introduces one or more mutually disjoint subsets of that set. The declaration on line 5 of Figure 4.3 introduces two sets named `A2` and `A3` that are subsets of `A`. For each subsignature, we introduce a predicate to represent set membership for that set, similar to the one created for top-level signatures and subset signatures. To relate `A2` and `A3` to `A`, we introduce the following formulas, similar to the one created for subset signature:

$$\forall x : A \bullet inA2(x) \Rightarrow inA(x)$$
$$\forall x : A \bullet inA3(x) \Rightarrow inA(x)$$

If there are multiple subsignatures at the same level in the set hierarchy, we include a formula that they must be disjoint:

$$\forall x : A \bullet \neg\ (inA2(x) \wedge inA3(x))$$

An *abstract* signature in Alloy has no elements besides those belonging to its extensions. Therefore, we add the formula in MSFOL that everything in the superset must be in one of the subsets. For the declarations on lines 7 to 9 in Figure 4.3, we include the formula:

$$\forall x : B \bullet inB(x) \Rightarrow (inB1(x) \vee inB2(x))$$

If a signature is not abstract, it is possible for it to have elements that are not in any of its extensions so the above formula is not added.

A signature declaration can also have a *multiplicity* associated with it. Set multiplicity in Alloy is expressed with a *multiplicity keyword* as in:

```
M sig A { }
```

where `M` can be any of:

- **one** : `A` has exactly one element.
- **lone** : `A` can have zero or one element.
- **some** : `A` can have one or more elements.

These multiplicity restrictions are equivalent to formulas of the form `M A`, which we show how to translate in Section 4.4.2. If a multiplicity keyword is not included in the signature declaration, `A` can have any number of elements.

Relations are declared as fields of signatures in Alloy. The declaration on line 2 of Figure 4.3 introduces a relation `f` whose domain is `A` and range is given by the expression `e`. The expression `e` can be a signature, a field or a combination of two or more signatures or fields joined by the operator `->`. In general, for any expression `e` of arity n in a relation declaration, we declare a predicate in MSFOL as:

$$f : sort_1(\texttt{A}) \times sort_1(\texttt{e}) \times \ldots \times sort_n(\texttt{e}) \rightarrow Bool$$

If any of the signatures in the type expression of `f` are subsignatures, they are lifted to their top-level signature according to the set hierarchy of the model (using our auxiliary function *sort*). Recall that MSFOL does not support a sort hierarchy.

In an Alloy instance, the tuples of a relation are constrained to contain elements from the corresponding sets in its type expression. For example, for relation `f: A1 -> B` where `A1` is the subsignature of the top-level signature `A`, `f` contains elements from $A \times B$ depending on the set membership of `A1` and `B`. In an instance, any elements not included

in `A1` cannot be in the domain of `f`. We need to restrict the values of each parameter in the function declaration. For relation `f: A -> e` where `e` can be any expression with arity n, we can express this constraint in Portus as:

$$\forall x_1 : A,\, x_2 : sort_1(\texttt{e}),\, \ldots,\, x_{n+1} : sort_n(\texttt{e}) \bullet$$
$$f(x_1, \ldots, x_{n+1}) \Rightarrow (inA(x_1) \wedge [\![(x_2, \ldots, x_{n+1}) \in \texttt{e}]\!])$$

The translation of Alloy formulas of the form $[\![(x_1, \ldots, x_n) \in \texttt{e}]\!]$ is described in Section 4.4.

The expression declaring a relation can be prefixed with a multiplicity keyword as in:

```
sig A { f: M e }
```

This multiplicity constraint is equivalent to the following Alloy fact:

```
all x : A | M x.f
```

Similarly, if `e` is an expression denoting a relation of arity two or more, it may contain multiplicity keywords within it. For instance, the declaration:

```
sig A { f : e1 M -> N e2 }
```

results in adding the translation of the following two Alloy formulas (in addition to the declaration of f):

```
all x : A | all y : e1 | N y.x.f
all x : A | all y : e2 | M x.f.y
```

where `e1` and `e2` can be Alloy expressions of any arity. The possible multiplicity keywords for `M` and `N` are `one`, `lone` or `some` with similar meanings as discussed above. We describe the translation of Alloy formulas in Section 4.4.

4.3 Step 2 - Translate scopes

The abstract syntax for the command declarations specifying a scope for signatures is included in Figure 4.4. In an Alloy command, finite scopes must be chosen for at least the top-level signatures[3]. The scopes can be exact or non-exact. For non-exact scopes, the analysis explores all instances where the number of elements in each signature can be any value between zero and the scope for that signature. The scope of abstract signatures can be expressed implicitly by assigning a scope to all of its subsignatures. The Alloy Analyzer passes the scope information for each signature to Portus including the signatures whose scope is derived implicitly or specified using multiplicity keywords. An error is thrown by the Alloy Analyzer if only some subsignatures have a scope assigned to them or if the scopes are conflicting, thus Portus does not need to handle these cases. In this section, we discuss how to handle non-exact and exact scopes of top-level signatures and subsignatures.

[3]If the user does not specify a scope, a default scope of 3 is assigned.

cmdDecl ::= [`run` | `check`] { formula* } [scope]
scope ::= `for` number [`but` typescope,+] | `for` typescope,+
typescope ::= [`exactly`] number sig

Figure 4.4: Abstract syntax for Alloy commands (taken from Jackson [31])
(Operators for this BNF notation are defined in Appendix B)

In Section 4.2, we discussed how PORTUS creates sorts for each top-level signature and membership predicates for every signature. These membership predicates not only allow for the expression of set hierarchy but these are key to handling non-exact scopes. The maximum scope of each top-level signature is passed to Fortress as the scope for the sort representing that signature. The number of elements that satisfy the membership predicate in the Fortress interpretation is equal to the number of elements in that signature in the Alloy instance (which may be less than the maximum scope size). If an exact scope is chosen for a top-level signature `A`, we add the following constraint to its membership predicate:

$$\forall x : A \bullet inA(x)$$

where A is the sort representing set `A`.

If a scope is specified for the subsignatures, we have to add extra constraints to ensure that these sets stay within their possible scope sizes. We enforce these limits by adding formulas on the set membership predicates. For example, for the declarations on lines 7 and 8 of Figure 4.3 on page 27, if the modeller chooses a scope of n for subsignature `B1`, we add the following formula to ensure that `B1` never contains more than n elements:

$$\begin{aligned}&\forall x_1 : B,\ \ldots,\ x_{n+1} : B \bullet \\ &\quad inB1(x_1) \wedge \ldots \wedge inB1(x_{n+1}) \Rightarrow \\ &\qquad (x_1 = x_2 \vee \ldots \vee x_1 = x_{n+1} \vee x_2 = x_3 \vee \ldots \vee x_2 = x_{n+1} \vee \ldots \vee x_n = x_{n+1})\end{aligned}$$

If an exact scope of n is chosen for subsignature `B1`, PORTUS translates it as:

$$\begin{aligned}
&\exists x_1 : B,\ \ldots,\ x_n : B \bullet \forall x_{n+1} : B \bullet \\
&\quad \neg(x_1 = x_2) \wedge \ldots \wedge \neg(x_1 = x_n) \wedge \\
&\quad \neg(x_2 = x_3) \wedge \ldots \wedge \neg(x_2 = x_n) \wedge \\
&\quad \ldots \wedge \neg(x_{n-1} = x_n) \wedge \\
&\quad (inB1(x_{n+1}) \iff (x_{n+1} = x_1 \vee \ldots \vee x_{n+1} = x_n))
\end{aligned}$$

The scope for integers in Alloy is specified in terms of bitwidth[4]. PORTUS retrieves the bitwidth information from the Alloy Analyzer and passes it to Fortress. If the modular aithmetic option is chosen for integers, Fortress uses this bitwidth to set the size of bit-vectors when finitizing integers.

4.4 Step 3 - Translate formulas

In order to translate an Alloy formula to MSFOL, the AST of the formula is traversed in a top-down order. The abstract syntax for formulas in the Alloy language is shown in Figure 4.5. Our translation operator $[\![\cdot]\!]$ is defined in this section. In our description of the translation, we divide the operators into subsections of similar operators. In some cases, we convert one Alloy formula into another equivalent Alloy formula. The only Alloy operators we do not cover are the bit shifting operators (`>>` and `<<`) which are rarely used in Alloy models.

To proceed in a top-down manner, we often have to pass down contextual information to the lower expression(s). This information may be broken down and passed by the lower expression to the expressions further down the tree. The result of $[\![\cdot]\!]$ is always a formula. For example, a translation of:

$$(x, y) \in \texttt{(e1 + e2)}$$

is converted to a translation of:

$$(x, y) \in \texttt{e1} \vee (x, y) \in \texttt{e2}$$

In the following subsections, we represent Alloy formulas using `f`, `f1` and `f2` and Alloy expressions with contextual information passed down by larger formulas using `e`, `e1` and `e2`.

[4]If the user does not specify a scope for integers, a default bitwidth of 3 is assigned.

formula ::= logFormula | quantFormula | elemFormula

logFormula ::= `not` formula | formula logOp formula
logOp ::= `and` | `or` | `implies` | `iff`

quantFormula ::= quantifier varDecl **|** formula
quantifier ::= `all` | `no` | mult
mult ::= `some` | `lone` | `one`
varDecl ::= var `:` expr

elemFormula ::= (mult | `no`) expr | expr compOp expr
compOp ::= `in` | `=` | `>` | `<` | `>=` | `<=`

expr ::= unOp expr | expr binOp expr | expr intOp expr |
sig | field | var | constant
unOp ::= `~` | `^` | `*` | `#`
binOp ::= `+` | `&` | `-` | `->` | `.` | `<:` | `:>` | `++`
intOp ::= `+` | `-` | `*` | `/` | `%` | `>>` | `<<`
sig ::= identifier
field ::= identifier `:` sig [`->` sig]$^+$
var ::= identifier
constant ::= `none` | `univ` | `iden`

Figure 4.5: Abstract syntax for Alloy formulas (taken from Jackson [31])
(Operators for this BNF notation are defined in Appendix B)

4.4.1 Logical Operators

Alloy uses the common logical operators to combine formulas into larger constraints. The straightforward translation of these operators into MSFOL is shown in Figure 4.6.

$$
\begin{aligned}
[\![\texttt{not f}]\!] &\coloneqq \neg[\![\texttt{f}]\!] \\
[\![\texttt{f1 and f2}]\!] &\coloneqq [\![\texttt{f1}]\!] \wedge [\![\texttt{f2}]\!] \\
[\![\texttt{f1 or f2}]\!] &\coloneqq [\![\texttt{f1}]\!] \vee [\![\texttt{f2}]\!] \\
[\![\texttt{f1 implies f2}]\!] &\coloneqq [\![\texttt{f1}]\!] \Rightarrow [\![\texttt{f2}]\!] \\
[\![\texttt{f1 iff f2}]\!] &\coloneqq [\![\texttt{f1}]\!] \Leftrightarrow [\![\texttt{f2}]\!]
\end{aligned}
$$

Figure 4.6: Translation of Logical Operators

$$
\begin{aligned}
[\![\texttt{all x : e | f}]\!] &\coloneqq \forall x : sort_1(\texttt{e}) \bullet [\![x \in \texttt{e}]\!] \Rightarrow [\![\texttt{f}]\!] \\
[\![\texttt{some x : e | f}]\!] &\coloneqq \exists x : sort_1(\texttt{e}) \bullet [\![x \in \texttt{e}]\!] \wedge [\![\texttt{f}]\!] \\
[\![\texttt{no x : e | f}]\!] &\coloneqq [\![\texttt{all x : e | not f}]\!] \\
[\![\texttt{lone x : e | f}]\!] &\coloneqq \forall x, x' : sort_1(\texttt{e}) \bullet \\
&\qquad [\![x \in \texttt{e}]\!] \wedge [\![x' \in \texttt{e}]\!] \wedge [\![\texttt{f}]\!] \wedge [\![\texttt{f}[x'/x]]\!] \Rightarrow x = x' \\
[\![\texttt{one x : e | f}]\!] &\coloneqq \exists x : sort_1(\texttt{e}) \bullet [\![x \in \texttt{e}]\!] \wedge [\![\texttt{f}]\!] \wedge \\
&\qquad \forall x' : sort_1(\texttt{e}) \bullet [\![x' \in \texttt{e}]\!] \wedge [\![\texttt{f}[x'/x]]\!] \Rightarrow x = x'
\end{aligned}
$$

Figure 4.7: Translation of Quantified Formulas
($arity(\texttt{e}) = 1$, $\texttt{f}[x'/x]$ is the formula `f` with all instances of x replaced with x')

4.4.2 Quantified Formulas and Expressions

A quantified formula in Alloy is of the form:

`Q x : e | f`

where `f` is a formula containing variable `x`, `e` is any expression bounding `x`, and `Q` is a quantifier. Alloy supports the following quantifiers: `all`, `some`, `no`, `lone`, `one`. The translation for formulas with quantified variables is shown in Figure 4.7 and is discussed below.

Quantified variables in first-order logic range over variables that represent a single element of a sort in the language but quantified variables in Alloy don't have to be scalars – they can be sets or relations. Although the Alloy language allows higher-order quantification, generally the Alloy Analyzer cannot handle such formulas unless higher-order quantifiers can be eliminated by skolemization. Since higher-order quantifications are rarely used in Alloy models, we chose not to include them in our translation.

The `all` quantifier means that `f` holds for every `x` in `e`. Since `e` can be any expression representing a set, we include a predicate to make sure `x` is a member of set `e` as antecedent of the translation of formula `f`. The `some` quantifier means that `f` holds for at least one

`x` in `e`. Similar to the quantifier `all`, we include the antecedent conjuncted with the translation of formula `f`. The quantifiers `no`, `lone` and `one` mean that the quantified formula is true for no, at most one, and exactly one `x` in `e` respectively and their translation is shown in Figure 4.7.

Quantifiers can be applied to expressions also. The Alloy expression:

```
M e
```

constrains the number of tuples in the relation `e` based on quantification `M`. The possible quantifiers are: `some`, `no`, `lone`, and `one` with similar meanings as discussed above. PORTUS translates these to MSFOL as shown in Figure 4.8. Since `e` may be a compound type in this case, it is decomposed to its basic sorts with a variable representing each tuple element.

$$
\begin{aligned}
[\![\texttt{some e}]\!] &\coloneqq \exists x_1 : sort_1(\texttt{e}), \ldots, x_n : sort_n(\texttt{e}) \bullet [\![(x_1, \ldots, x_n) \in \texttt{e}]\!] \\
[\![\texttt{no e}]\!] &\coloneqq \forall x_1 : sort_1(\texttt{e}), \ldots, x_n : sort_n(\texttt{e}) \bullet \neg [\![(x_1, \ldots, x_n) \in \texttt{e}]\!] \\
[\![\texttt{lone e}]\!] &\coloneqq \forall x_1 : sort_1(\texttt{e}), \ldots, x_n : sort_n(\texttt{e}) \bullet [\![(x_1, \ldots, x_n) \in \texttt{e}]\!] \Rightarrow \\
&\qquad \forall y_1 : sort_1(\texttt{e}), \ldots, y_n : sort_n(\texttt{e}) \bullet \\
&\qquad\quad [\![(y_1, \ldots, y_n) \in \texttt{e}]\!] \Rightarrow x_1 = y_1 \land \ldots \land x_n = y_n \\
[\![\texttt{one e}]\!] &\coloneqq \exists x_1 : sort_1(\texttt{e}), \ldots, x_n : sort_n(\texttt{e}) \bullet [\![(x_1, \ldots, x_n) \in \texttt{e}]\!] \land \\
&\qquad \forall y_1 : sort_1(\texttt{e}), \ldots, y_n : sort_n(\texttt{e}) \bullet \\
&\qquad\quad [\![(y_1, \ldots, y_n) \in \texttt{e}]\!] \Rightarrow x_1 = y_1 \land \ldots \land x_n = y_n
\end{aligned}
$$

Figure 4.8: Translation of Quantified Expressions ($arity(\texttt{e}) = n$)

4.4.3 Set Predicates

There are two predicates on sets in Alloy: subset (`e1 in e2`) and equality (`e1 = e2`). These are translated to MSFOL as shown in Figure 4.9. Because there are no scalars in Alloy, `in` is used to denote set membership between an expression that looks like a scalar (e.g. a quantified variable) and a set in the conventional sense and also to express the subset relationship between two sets or relations. The operands to these two binary predicates must be of equivalent arity.

4.4.4 Set Operators

Set operators produce sets and thus are not an Alloy formula themselves – they are always part of a larger formula. To properly translate a set operator in an expression, such

$$
\begin{aligned}
[\![\texttt{e1 in e2}]\!] &\coloneqq \forall x_1 : sort_1(\texttt{e1}), \ldots, x_n : sort_n(\texttt{e1}) \bullet \\
&\quad [\![(x_1, \ldots, x_n) \in \texttt{e1}]\!] \Rightarrow [\![(x_1, \ldots, x_n) \in \texttt{e2}]\!] \\
[\![\texttt{e1 = e2}]\!] &\coloneqq \forall x_1 : sort_1(\texttt{e1}), \ldots, x_n : sort_n(\texttt{e1}) \bullet \\
&\quad [\![(x_1, \ldots, x_n) \in \texttt{e1}]\!] \Leftrightarrow [\![(x_1, \ldots, x_n) \in \texttt{e2}]\!]
\end{aligned}
$$

Figure 4.9: Translation of Set Predicates ($arity(\texttt{e1}) = arity(\texttt{e2}) = n$)

as set union (`e1 + e2`), the top-down translation passes down contextual information regarding the tuple that must be in the set so that the expression can be decomposed into its components as individual formulas. Therefore, instead of translating `e` directly, we translate $(x_1, \ldots, x_n) \in \texttt{e}$ where $arity(\texttt{e}) = n$. The translation for the set operators union (`e1 + e2`), intersection (`e1 & e2`), and difference (`e1 - e2`) are shown in Figure 4.10 where `e1` and `e2` have equivalent arities.

4.4.5 Relational Operators

Similar to set operators, relational operators are always part of a larger Alloy formula. Therefore, instead of translating `e` directly, we translate $(x_1, \ldots, x_n) \in \texttt{e}$ where $arity(\texttt{e}) = n$. The translation of the relational operators is shown in Figure 4.11.

Alloy's relational product operator (`e1 -> e2`) results in the relation constructed by taking the product of all tuples in `e1` with all tuples in `e2` where `e1` and `e2` can have different arities.

The join operator in Alloy (`e1 . e2`) is the relation constructed by taking the join of each tuple of `e1` with each tuple of `e2`. Taking the join of a pair of tuples

$$(x_1, \ldots, x_n) \ . \ (y_1, \ldots, y_m)$$

is equal to

$$(x_1, \ldots, x_{n-1}, y_2, \ldots, y_m)$$

if $x_n = y_1$ and empty otherwise. `e1` and `e2` can have different arities but at least one of them needs to have an arity of more than one.

The transpose operator in Alloy (`~`) is a unary operator on a binary relation that reverses the order of the elements in the pairs of the relation.

$$
\begin{array}{rcl}
[\![(x_1,\ldots,x_n) \in \texttt{(e1 + e2)}]\!] & \coloneqq & [\![(x_1,\ldots,x_n) \in \texttt{e1}]\!] \vee [\![(x_1,\ldots,x_n) \in \texttt{e2}]\!] \\
[\![(x_1,\ldots,x_n) \in \texttt{(e1 \& e2)}]\!] & \coloneqq & [\![(x_1,\ldots,x_n) \in \texttt{e1}]\!] \wedge [\![(x_1,\ldots,x_n) \in \texttt{e2}]\!] \\
[\![(x_1,\ldots,x_n) \in \texttt{(e1 - e2)}]\!] & \coloneqq & [\![(x_1,\ldots,x_n) \in \texttt{e1}]\!] \wedge \neg[\![(x_1,\ldots,x_n) \in \texttt{e2}]\!]
\end{array}
$$

Figure 4.10: Translation of Set Operators ($arity(\texttt{e1}) = arity(\texttt{e2}) = n$)

$$
\begin{array}{rcl}
[\![(x_1,\ldots,x_n) \in \texttt{(e1 -> e2)}]\!] & \coloneqq & [\![(x_1,\ldots,x_m) \in \texttt{e1}]\!] \wedge [\![(x_{m+1},\ldots,x_n) \in \texttt{e2}]\!] \\
& & \text{where } m < n,\ arity(\texttt{e1}) = m \text{ and} \\
& & arity(\texttt{e2}) = n - m. \\
[\![(x_1,\ldots,x_n) \in \texttt{(e1 . e2)}]\!] & \coloneqq & \exists y : sort_1(\texttt{e2}) \bullet [\![(x_1,\ldots,x_m,y) \in \texttt{e1}]\!] \wedge \\
& & \quad [\![(y,x_{m+1}\ldots,x_n) \in \texttt{e2}]\!] \\
& & \text{where } m < n,\ arity(\texttt{e1}) = m + 1 \text{ and} \\
& & arity(\texttt{e2}) = n - m + 1. \\
[\![(x_1,x_2) \in \sim\texttt{e}]\!] & \coloneqq & [\![(x_2,x_1) \in \texttt{e}]\!] \text{ where } arity(\texttt{e}) = 2. \\
[\![(x_1,x_2) \in \texttt{\^{}e}]\!] & \coloneqq & [\![(x_1,x_2) \in \texttt{(e + e}^2\texttt{ + ... +e}^k\texttt{)}]\!] \\
[\![(x_1,x_2) \in \texttt{*e}]\!] & \coloneqq & [\![(x_1,x_2) \in \texttt{(\^{}e + iden)}]\!] \\
& & \text{where } arity(\texttt{e}) = 2,\ scope(sort_1(\text{e})) = k \text{ and} \\
& & \texttt{e}^i \text{ is the join of e } i \text{ times.} \\
[\![(x_1,\ldots,x_n) \in \texttt{(e1 <: e2)}]\!] & \coloneqq & [\![x_1 \in \texttt{e1}]\!] \wedge [\![(x_1,\ldots,x_n) \in \texttt{e2}]\!] \\
& & \text{where } arity(\texttt{e1}) = 1 \text{ and } arity(\texttt{e2}) = n. \\
[\![(x_1,\ldots,x_n) \in \texttt{(e1 :> e2)}]\!] & \coloneqq & [\![(x_1,\ldots,x_n) \in \texttt{e1}]\!] \wedge [\![x_n \in \texttt{e2}]\!] \\
& & \text{where } arity(\texttt{e1}) = n \text{ and } arity(\texttt{e2}) = 1. \\
[\![(x_1,\ldots,x_n) \in \texttt{(e1 ++ e2)}]\!] & \coloneqq & [\![(x_1,\ldots,x_n) \in \texttt{e2}]\!] \vee \\
& & \neg\ (\ \exists y_2 : sort_2(\texttt{e2}), \ldots, y_n : sort_n(\texttt{e2}) \bullet \\
& & \quad [\![(x_1,y_2,\ldots,y_n) \in \texttt{e2}]\!]\ ) \wedge \\
& & [\![(x_1,\ldots,x_n) \in \texttt{e1}]\!]\ ) \\
& & \text{where } arity(\texttt{e1}) = arity(\texttt{e2}) = n.
\end{array}
$$

Figure 4.11: Translation of Relational Operators

A binary relation is *transitive* if, whenever it contains the tuples `a -> b` and `b -> c`, it also contains `a -> c`. The *transitive closure* `^r` of a binary relation `r` is the smallest relation that contains `r` and is transitive. It can be computed as:

```
^r = r + r.r + r.r.r + ...
```

A binary relation is *reflexive* if it contains the tuple `a -> a` for every element `a`. The *reflexive transitive closure* `*r` of a binary relation `r` is the smallest relation that contains `r` and is both transitive and reflexive. It can be computed as:

```
*r = ^r + iden
```

The closure operators are not axiomatizable in first-order logic unless the domain is finite. For a scope of k, the computation of closure converges in at most k steps [29]. For instance, for a scope of 3 for Alloy expression `e` of arity 2, we can translate closure as:

$$[\![(x_1, x_2) \in \texttt{\^{}e}]\!] \coloneqq [\![(x_1, x_2) \in \texttt{(e + e.e + e.e.e)}]\!]$$

Transitive closure is one of the operators where the scope of a set is used in the basic translation (In the optimizations, where we move to axiomatized versions of transitive closure for finite sets, we do not require the scope in the translation.). Other approaches to translate closure operators are discussed in detail and compared to each other in Section 5.7.

The restriction operators in Alloy are used to filter relations to a given domain or range. The domain restriction operator (`e1 <: e2`) contains all tuples in relation `e2` which start with an element in set `e1`. Similarly, the range restriction operator (`e1 :> e2`) contains all tuples in relation `e1` which end with an element in set `e2`.

Alloy's override (`e1 ++ e2`) operator is similar to the union of `e1` and `e2` except that tuples in `e2` replace tuples in `e1` if they start with the same element.

4.4.6 Leaf Expressions

All the previous translations eventually reduce to a translation of an Alloy formula of the form $(x_1, ..., x_n) \in \texttt{R}$ as shown in Figure 4.12 where `R` can be an Alloy signature, field, variable or constant.

For the case of $x \in \texttt{A}$ where `A` is a signature in Alloy, it is translated directly to the corresponding set membership predicate inA in MSFOL. For $(x_1, ..., x_n) \in \texttt{R}$, where `R` is a relation in Alloy, we use the corresponding predicate R in MSFOL. For a variable `v` in Alloy, we can translate $x \in \texttt{v}$ as $x = v$ where $[\![\texttt{v}]\!] = v$.

Alloy has three constants: `none`, `univ` and `iden`. `none` and `univ` are the sets containing no and all elements respectively. The Alloy constant `iden` represents the

identity binary relation over the entire universe. However, `iden` is usually restricted in some way to take the identity of a particular set in Alloy, not the entire universe, as in:

```
A <: iden
```

PORTUS translates $(x_1, x_2) \in$ `iden` as $x_1 = x_2$. PORTUS's translation for `iden` takes into account the relevant elements based on the context of the formula. Since Kodkod is untyped and Fortress is sorted, one of the advantages of using PORTUS over Kodkod in Alloy is that sort information about the elements in `iden` is retained as the Alloy formula is traversed in a top-down manner and the restriction operators are not needed. For instance, the constraint

```
iden in f
```

to check if the relation `f` is reflexive[5] would be inconsistent in Alloy when using Kodkod as back-and but when using PORTUS, this is translated to:

$$\forall x_1 : sort_1(\texttt{f}),\ x_2 : sort_2(\texttt{f}) \bullet x_1 = x_2 \Rightarrow f(x_1, x_2)$$

which is exactly what the user had in mind when writing that constraint. Using this approach, both the Alloy expressions `A <: iden in f` and `A in f` translate to an equivalent meaning in PORTUS.

$$\begin{aligned}
[\![x \in \texttt{A}]\!] &\coloneqq inA(x) \\
[\![(x_1, \ldots, x_n) \in \texttt{f}]\!] &\coloneqq f(x_1, \ldots, x_n) \\
[\![x \in \texttt{v}]\!] &\coloneqq x = v \\
\\
[\![x \in \texttt{none}]\!] &\coloneqq \bot \\
[\![x \in \texttt{univ}]\!] &\coloneqq \top \\
[\![(x_1, x_2) \in \texttt{iden}]\!] &\coloneqq x_1 = x_2
\end{aligned}$$

Figure 4.12: Translation of Leaf Expressions

[5] This example is taken from [31] when mentioning the drawbacks of forgetting to apply the restriction operator on `iden`.

4.4.7 Integer Operators

The only non second-order built-in operators in Alloy besides logical, set and relational operators are ones for integers. There are two possible semantics for finite-size integers in Alloy: modular arithmetic and no overflow. Modular arithmetic results in wraparound in case of overflow but can lead to spurious counterexamples. No overflow semantics were introduced to suppress instances which result in overflows [42]. PORTUS uses the modular arithmetic option for integers in Fortress which corresponds to the modular arithmetic semantics in Alloy. Currently PORTUS does not provide support for the no overflow semantics in Alloy.

Since Fortress provides support for integers as a built-in sort, PORTUS translates Alloy integer operators to their equivalent integer operators in Fortress as shown in Figure 4.13. Recall that if the modular arithmetic option for integers is chosen in Fortress, all integer operators are converted to their equivalent bit-vector operators i.e. for a bitwidth of n, $>$ is converted to $>_{[n]}$, $+$ is converted to $+_{[n]}$ and so on.

$$
\begin{array}{rcl}
[\![\texttt{e1 = e2}]\!] & := & [\![\texttt{e1}]\!] = [\![\texttt{e2}]\!] \\
[\![\texttt{e1 < e2}]\!] & := & [\![\texttt{e1}]\!] < [\![\texttt{e2}]\!] \\
[\![\texttt{e1 > e2}]\!] & := & [\![\texttt{e1}]\!] > [\![\texttt{e2}]\!] \\
[\![\texttt{e1 <= e2}]\!] & := & [\![\texttt{e1}]\!] \geq [\![\texttt{e2}]\!] \\
[\![\texttt{e1 >= e2}]\!] & := & [\![\texttt{e1}]\!] \leq [\![\texttt{e2}]\!]
\end{array}
\qquad
\begin{array}{rcl}
[\![\texttt{e1 + e2}]\!] & := & [\![\texttt{e1}]\!] + [\![\texttt{e2}]\!] \\
[\![\texttt{e1 - e2}]\!] & := & [\![\texttt{e1}]\!] - [\![\texttt{e2}]\!] \\
[\![\texttt{e1 * e2}]\!] & := & [\![\texttt{e1}]\!] * [\![\texttt{e2}]\!] \\
[\![\texttt{e1 / e2}]\!] & := & [\![\texttt{e1}]\!] \,/\, [\![\texttt{e2}]\!] \\
[\![\texttt{e1 \% e2}]\!] & := & [\![\texttt{e1}]\!] \,\%\, [\![\texttt{e2}]\!]
\end{array}
$$

Figure 4.13: Translation of Integer Operators

The *set cardinality* operator `#` returns the number of elements contained in a set as an integer value. PORTUS introduces a novel approach to handle set cardinality using the built-in sort for integers. For the translation of `#e` where the Alloy expression `e` has arity n, we introduce an auxiliary function:

$$countR : sort_1(\texttt{e}) \times \ldots \times sort_n(\texttt{e}) \rightarrow Int$$

and a formula limiting $countR$:

$$
\begin{aligned}
&\forall x_1 : sort_1(\texttt{e}) , \ldots , x_n : sort_n(\texttt{e}) \bullet \\
&\quad (R(x_1, \ldots, x_n) \Rightarrow countR(x_1, \ldots, x_n) = 1) \wedge \\
&\quad (\neg R(x_1, \ldots, x_n) \Rightarrow countR(x_1, \ldots, x_n) = 0)
\end{aligned}
$$

where each tuple in the relation `e` is mapped to an integer value in $countR$. As a result, the cardinality of `e` can be determined by adding all the values in $countR$.

Given an Alloy expression `e` of the type expression `A -> B` where $scope(\texttt{A}) = a$ and $scope(\texttt{B}) = b$, the set cardinality operator is translated as:

$$\begin{aligned}[\![\texttt{\#e}]\!] \coloneqq\ & countR(1_A, 1_B) + countR(2_A, 1_B) + \ldots + countR(a_A, 1_B) + \\ & countR(1_A, 2_B) + countR(2_A, 2_B) + \ldots + countR(a_A, 2_B) + \\ & \ldots \\ & countR(1_A, b_B) + countR(2_A, b_B) + \ldots + countR(a_A, b_B)\end{aligned}$$

4.4.8 Modules

The Alloy language allows a modeller to split their model into several *modules*, and import these into a main module. A module may take sets as parameters and add specific relations and assertions on its parameters to the model. Within the Analyzer, these modules are loaded and relevant substitutions are performed prior to our translation step, so our process does not need any special support for modules. Alloy also provides built-in modules for common operations including the ordering module and modules for integers, booleans, binary and ternary relations. The built-in modules are treated the same as user-defined modules and translated as normal formulas. The ordering module is discussed in more detail later in Section 5.6.

4.5 Step 4 - Solve using Fortress

After translating the Alloy problem to an equivalent MSFMF problem, Portus invokes an appropriate model finder from Fortress to check the satisfiability of the MSFOL formulas. The modular arithmetic option for integers is chosen in Fortress. The model finder from Fortress translates the given problem to the logic of EUFBV through a series of transformations as discussed in Section 2.3 and feeds the resulting formulas to an SMT solver.

4.6 Step 5 - Return instances

If the MSFMF problem is satisfiable, Fortress returns interpretations for all constant, function and predicate symbols in addition to the domain for each sort. Portus converts this interpretation into an Alloy instance. Portus uses the set membership predicates,

as defined in Section 4.2, to determine the membership of the basic sets and subsets. It uses the predicates representing Alloy relations to determine the tuples in each relation. In addition, interpretations for skolem constants or functions introduced during skolemization in Fortress are mapped back to corresponding skolem constants or functions in Alloy.

To utilize the existing infrastructure of the Alloy code base, PORTUS converts the Fortress interpretation to an equivalent Kodkod instance so Alloy can process the solution the same way it would have done if any of the SAT solvers were used for analysis. Because Kodkod is a lower-level language than Alloy, PORTUS first defines the mapping from all Fortress domain elements to Kodkod elements, finds the appropriate Kodkod relation for each signature and field and finally, adds the relevant tuples for each Kodkod relation to the solution instance. The correctness of this mapping is determined by testing and the details are mentioned in Chapter 6.

4.7 Summary

In this chapter, we introduced the basic rules for translating an Alloy model to an MSFMF problem in PORTUS with each step discussed in detail. We cover all Alloy constructs except the bit-shifting operators and higher-order quantification. We present a novel method to translate the set cardinality operator using the built-in sort for integers. Since there is more than one way to translate an Alloy model, there is room for optimization in certain cases, which will be discussed in the next chapter.

Chapter 5

Optimizations

This chapter identifies opportunities for optimizations in the approach discussed in the previous chapter. For each optimization, we start by specifying the criteria needed to apply the optimization and then describe the optimized translation. Keeping the translation in the previous chapter as baseline, we evaluate the effect of each optimization on Alloy models using:

- the performance of Portus, and
- the profiling characteristics of the generated MSFMF problem.

Each optimization is tested using three Alloy models with increasing scopes. The Alloy models used for analysis in this chapter[1] are taken either from the models provided with the Alloy Analyzer or models used for evaluation in the research papers related to Alloy [27, 31, 41, 57]. The models are chosen to isolate the effect of the optimization as much as possible. The scopes are set to be non-exact unless stated otherwise. Each model has only one `check` command and is unsatisfiable, unless stated otherwise. Unsatisfiable models and non-exact scopes are better for the evaluation of any optimization because they are usually harder to analyze since the possible number of instances to check are greater. All experiments were run on Intel®Xeon®CPU E3-1240 v5 @ 3.50 GHz with Ubuntu 16.04 64-bit with up to 64GB of user memory. We repeat each experiment five times and report the average with a time limit of 30 minutes on each process. Information about the version of each tool we used is available in Appendix A.

Section 2.3 mentioned the five steps performed by Fortress to translate a problem to EUFBV: 1) finitize integers 2) normalize 3) ground formulas 4) add range formulas and 5)

[1] Available at https://github.com/WatForm/portus-tests.

add domain element formulas. Keeping the complexity of each step in mind, we apply a combination of one or more of the following in each optimization:

- Remove a predicate resulting in fewer range formulas.
- Replace an n-ary predicate with a function of arity $n - 1$ or replace a unary predicate with a constant resulting in fewer and smaller range formulas. Constants and functions also have better symmetry breaking techniques as compared to predicates.
- Remove an existentially quantified variable resulting in fewer skolem constants or functions created during normalization and consequently, fewer range formulas added.
- Remove a universally quantified variable so grounding formulas takes exponentially less time.
- Add symmetry breaking for a predicate.

As mentioned above, in addition to performance, we also compare certain profiling characteristics of the MSFMF problem created by Portus for each model. These characteristics include:

- the number of sorts, function, predicate and constant symbols in the language of the MSFMF problem,
- the maximum arity of function and predicate symbols in the language of the MSFMF problem,
- the maximum depth of quantification in the set of MSFOL formulas in the MSFMF problem,
- the number of skolem constant and function symbols created during skolemization by Fortress,
- the total number of terms in the set of MSFOL formulas in the MSFMF problem before and after conversion to EUFBV logic, and
- the total time in seconds taken by Fortress to convert the MSFMF problem to EUFBV logic, which we call the *transform time*.

We only report the characteristics that are affected for each optimization. Since the term count is very large for all models, we instead report the factor by which the total term count changes after applying the optimized translation.

5.1 Optimization of Join

As discussed in Section 4.4.5, the join operator `(e1 . e2)` in Alloy is translated to the MSFOL formula:

$$[\![(x_1, ..., x_n) \in \texttt{e1 . e2}]\!] \coloneqq \exists y : sort_1(\texttt{e2}) \bullet [\![(x_1, \ldots, x_m, y) \in \texttt{e1}]\!] \wedge [\![(y, x_{m+1}, \ldots, x_n) \in \texttt{e2}]\!]$$

where $m < n$, $arity(\texttt{e1}) = m+1$ and $arity(\texttt{e2}) = n-m+1$. However, if we know that one of `e1` or `e2` is an Alloy variable, this translation can be simplified to a formula without quantifiers.

5.1.1 Change in Step 3 - Translate Formulas

If `e1` is an Alloy variable, it is translated to a variable in MSFOL and we can use this variable directly in our translation of the join operator as in:

$$[\![(x_1, ..., x_n) \in \texttt{(e1 . e2)}]\!] \coloneqq [\![(v, x_1, ..., x_n) \in \texttt{e2}]\!]$$

where $[\![\texttt{e1}]\!] = v$ and $arity(\texttt{e1 . e2}) = arity(\texttt{e2}) - 1 = n$. Similarly, if `e2` is an Alloy variable:

$$[\![(x_1, ..., x_n) \in \texttt{(e1 . e2)}]\!] \coloneqq [\![(x_1, ..., x_n, v) \in \texttt{e1}]\!]$$

where $[\![\texttt{e2}]\!] = v$ and $arity(\texttt{e1 . e2}) = arity(\texttt{e1}) - 1 = n$. `e1` and `e2` both cannot be variables.

5.1.2 Evaluation

The join operator is one of the most common operators used in Alloy models. The results of applying the optimization of the join operator are demonstrated in Table 5.1. The `ceilingsAndFloors` model contains simple joins of a variable and a relation. The `lights` and `addressBook1h` models contain nested joins of variables and the `addressBook1h` model also contains a set heirarchy.

	ceilingsAndFloors			lights			addressBook1h		
Scope	7	8	9	20	30	40	12	15	18
w/o opt	8.68	25.75	593.42	4.54	18.25	56.56	3.82	13.11	43.11
w opt	0.43	1.13	3.15	0.34	0.71	1.18	0.39	0.64	1.08

(a) Performance Results (in seconds)

	ceilingsAndFloors		lights		addressBook1h	
	Scope = 8		Scope = 30		Scope = 15	
	w/o opt	w opt	w/o opt	w opt	w/o opt	w opt
Quant Depth	4	3	8	6	8	7
Skolem Constants	1	1	10	8	9	7
Skolem Functions	14	6	21	6	4	0
Initial Term Count	×1	×0.68	×1	×0.66	×1	×0.64
Final Term Count	×1	×0.38	×1	×0.05	×1	×0.57
Transform Time (s)	0.02	0.02	15.38	0.12	12.10	0.18

(b) Profiling Characteristics

Table 5.1: Optimization of the join operator in Portus

Based on the results in Table 5.1a, this optimization has a huge impact on solving time and scales well with increasing scopes. Table 5.1b presents the profiling characteristics for one of the scopes of each Alloy model. As predicted, the join optimization results in less quantifier depth and less number of skolem constants and functions. This effect is particularly noticeable when comparing the term counts before and after translation to EUFBV. Another observation is the transform time for the models `addressBook1h` and `lights` is equal to 92% and 84% of the solving time respectively. Since the join operator is used quite frequently in Alloy models and the unoptimized translation makes the analysis infeasible even on small scopes because of its effect on the transform time for Fortress, we choose to include this optimization to our baseline (and to the optimized translation) when comparing against further optimizations.

5.2 Optimization of Exact Scopes

As discussed in Section 4.2, Portus translates top-level signatures to sorts in MSFOL along with a predicate for set membership to handle non-exact scopes. If an exact scope is chosen for a top-level signature, the membership predicate for the top-level signature can be removed since it is not needed anymore. In addition, the formulas for each step in the translation process can be simplified in certain cases as mentioned below.

5.2.1 Change in Step 1 - Translate signatures

In addition to the membership predicate being removed for top-level signatures with exact scopes, the constraints on any subsets or fields associated with that top-level signature are also simplified. The constraint to relate a subset signature or subsignature `A1` to the top-level signature `A`:

$$\forall x : A \bullet inA1(x) \Rightarrow inA(x)$$

is removed if the scope of `A` is of an exact size. The translation of an abstract top-level signature `A` with an exact scope and subsignatures `A1` and `A2`:

$$\forall x : A \bullet inA(x) \Rightarrow inA1(x) \lor inA2(x)$$

is simplified to:

$$\forall x : A \bullet inA1(x) \lor inA2(x)$$

where $inA1$ and $inA2$ are the membership predicates for `A1` and `A2`. Similarly, the formula to relate a field `f` to its type expression `A -> e`:

$$\begin{aligned}\forall x_1 : A,\, x_2 : sort_1(\mathtt{e}),\, \ldots,\, x_{n+1} : sort_n(\mathtt{e}) \bullet \\ f(x_1, \ldots, x_{n+1}) \Rightarrow (inA(x_1) \land [\![(x_2, \ldots, x_{n+1}) \in \mathtt{e}]\!])\end{aligned}$$

is simplified to:

$$\forall x_1 : A, x_2 : sort_1(\mathtt{e}) \,, \ldots, x_{n+1} : sort_n(\mathtt{e}) \bullet f(x_1, \ldots, x_{n+1}) \Rightarrow [\![(x_2, \ldots, x_{n+1}) \in \mathtt{e}]\!]$$

if `A` is a top-level signature with an exact scope.

5.2.2 Change in Step 2 - Translate scopes

The constraints for exact scopes are removed for top-level signatures.

5.2.3 Change in Step 3 - Translate formulas

The translation for the leaf expression for a top-level signature `A`:

$$[\![x \in \texttt{A}]\!] \;\coloneqq\; inA(x)$$

is modified to:

$$[\![x \in \texttt{A}]\!] \;\coloneqq\; \top$$

if `A` is a top-level signature with an exact scope. The translation for the quantifier formulas of the form:

```
Q x : e | f
```

where the quantifier `Q` is equal to `all`, `some`, `lone` or `one` is simplified if `e` is a top-level signature with an exact scope. The antecedent $[\![x \in \texttt{e}]\!]$ is removed for the quantifiers mentioned above since it is true for all values of x.

5.2.4 Change in Step 5 - Return instances

The instance for the top-level signature is obtained by including all elements mapped to the corresponding sort in the domain assignment.

5.2.5 Evaluation

Exact scopes are quite common in Alloy models. The use of the ordering module on a signature adds an implicit constraint for that signature to have an exact scope. Table 5.2 demonstrates the effect of removing the membership predicate for top-level signatures on the performance of Portus. The models `filesystem`, `lights` and `addressBook1h` have one, three and three signatures with scopes of an exact size respectively and the `filesystem` and `addressBook1h` models contain a set hierarchy. All models are assigned exact scopes as is required by the optimization criteria.

	filesystem			lights			addressBook1h		
Scope	16	18	20	45	65	85	25	35	45
w/o opt	13.44	50.38	78.22	1.99	5.96	12.45	3.34	21.05	114.85
w opt	9.89	33.04	157.98	1.62	4.51	11.53	3.93	15.75	55.47

(a) Performance Results (in seconds)

	filesystem		lights		addressBook1h	
	Scope = 18		Scope = 65		Scope = 35	
	w/o opt	w opt	w/o opt	w opt	w/o opt	w opt
Predicates	5	4	9	6	4	1
Initial Term Count	×1	×0.98	×1	×0.89	×1	×0.68
Final Term Count	×1	×0.99	×1	×0.89	×1	×0.93
Transform Time (s)	0.09	0.09	0.43	0.37	2.65	2.46

(b) Profiling Characteristics

Table 5.2: Optimization of exact scopes on top-level signatures in Portus

The results in Table 5.2a demonstrate that removing the membership predicate and simplifying formulas results in a slight improvement in performance for these models. Table 5.2b shows the number of predicates removed due to exact scopes for each model and the effect on term counts before and after translation to EUFBV. One unexpected result is the performance of the model `filesystem` for scope 20 with the optimization where the performance was much worse. We repeated this experiment multiple times but the same result was observed which is attributed to the SMT solver's unpredictable behavior.

5.3 Optimization of Signature Hierarchy

For each signature, Portus introduces a predicate for set membership as discussed in Section 4.2. Given an abstract, top-level signature `A` in Alloy with an exact scope and exactly two subsignatures, `A1` and `A2`, we can remove the membership predicate for one of the subsignatures. If the predicate for `A2` is removed, its membership can be deduced by the expression `A - A1` i.e. all elements included in set `A` and not included in set `A1`.

5.3.1 Change in Step 1 - Translate signatures

In addition to removing the membership predicate for one of the subsignatures, the constraints to ensure that the subsignatures are disjoint and every element in the superset is a part of one of the subsets are removed.

5.3.2 Change in Step 3 - Translate formulas

The translation for the leaf expression:

$$[\![x \in \texttt{A2}]\!] \;\coloneqq\; inA2(x)$$

is modified to:

$$[\![x \in \texttt{A2}]\!] \;\coloneqq\; \neg\, inA1(x)$$

where `A1` and `A2` are subsignatures and membership predicate for `A2` is removed.

5.3.3 Change in Step 5 - Return instances

Portus uses the membership of the parent signature and the subsignature to determine the elements belonging to the subsignature with its membership predicate removed.

5.3.4 Evaluation

Table 5.3 demonstrates the effect of removing the membership predicate for one of the subsignatures of a top-level signature of an exact scope. All models are assigned exact scopes as is required by the optimization criteria. Although the effect on term count is not that significant as shown in Table 5.3b, Table 5.3a shows that 5 out of 9 models have improved performance, particularly noticeable for the last model, `grandpa`, where the solver times out for the unoptimized translation. Unexpected results are observed for the `addressBook2e` model where increasing the scope does not have the predicted effect and larger scopes take less time to solve than smaller scopes. These results are attributed to the SMT solver's unpredictable behavior.

	addressBook2e			filesystem			grandpa		
Scope	9	11	13	12	15	18	16	22	28
w/o opt	1.53	228.65	15.71	10.35	35.89	114.76	1.06	929.64	t/o
w opt	68.77	32.04	13.66	10.25	41.70	123.60	3.35	3.13	7.31

(a) Performance Results (in seconds)
(t/o = timed out after 30 minutes)

	addressBook2e		filesystem		grandpa	
	Scope = 11		Scope = 15		Scope = 22	
	w/o opt	w opt	w/o opt	w opt	w/o opt	w opt
Predicates	12	11	9	8	12	11
Initial Term Count	×1	×0.95	×1	×0.87	×1	×0.95
Final Term Count	×1	×1	×1	×1	×1	×1
Transform Time (s)	0.38	0.37	0.07	0.07	0.47	0.35

(b) Profiling Characteristics

Table 5.3: Optimization of signature heirarchy in Portus

5.4 Optimization of Signatures

For each signature, Portus introduces a predicate for set membership as mentioned in Section 4.2. If the scope for a signature is set to one and to be of an exact size, we can replace the set membership predicate with a constant. Another way to specify such a scope is by adding the multiplicity `one` before the signature declaration:

```
one sig A { }
```

5.4.1 Change in Step 1 - Translate signatures

A top-level signature `A` with a scope of exactly one is translated to a sort A with a scope of one and a constant symbol $c_A \in A$. A subset signature `A1` with a scope of exactly one is translated to a constant $c_{A1} \in A$ where $sort_1(\texttt{A}) = A$ and `A` is the parent signature of `A1`. The formula to relate `A` and `A1`:

$$\forall x : A \bullet inA1(x) \Rightarrow inA(x)$$

is modified to:

$$inA(c_{A1})$$

where inA is the membership predicate for `A`. Similarly, a subsignature `A2` (of `A`) with a scope of exactly one is translated to a constant $c_{A2} \in A$ and related to `A` by the formula:

$$inA(c_{A2})$$

If there is another subsignature `A3` at the same level in the set heirarchy as `A2`, the constraint added to ensure that they are disjoint:

$$\forall x : A \bullet \neg(inA2(x) \land inA3(x))$$

is modified to:

$$\neg\ inA3(c_{A2})$$

if set `A3` is translated to a membership predicate, or:

$$c_{A2} \neq c_{A3}$$

if `A3` has a scope of exactly one and is translated to a constant $c_{A3} \in A$. If an abstract signature `B` has subsignatures `B1` and `B2` with `B1` having a scope of exactly one, the formula to relate `B1` and `B2` to `B`:

$$\forall x : B \bullet inB(x) \Rightarrow inB1(x) \lor inB2(x)$$

is modified appropriately:

$$\forall x : B \bullet inB(x) \Rightarrow x = c_{B1} \lor inB2(x)$$

5.4.2 Change in Step 2 - Translate scopes

The constraint for exact scopes is removed for top-level signatures, subset signatures and subsignatures if any of them are translated to a constant.

5.4.3 Change in Step 3 - Translate formulas

The translation of the leaf expression for set memebership:

$$[\![x \in \texttt{A}]\!] \coloneqq inA(x)$$

is modified to:

$$[\![x \in \texttt{A}]\!] \coloneqq x = c_A$$

The translation of the subset predicate:

$$[\![\texttt{e1 in e2}]\!] \coloneqq \forall x : sort_1(\texttt{e1}) \bullet [\![x \in \texttt{e1}]\!] \Rightarrow [\![x \in \texttt{e2}]\!]$$

is modified to:

$$[\![\texttt{e1 in e2}]\!] \coloneqq [\![c_{e1} \in \texttt{e2}]\!]$$

where `e1` is a signature representing a constant c_{e1} in MSFOL, thus, removing a universally quantified variable. The join operator is simplified using the optimization discussed in Section 5.1 if one of the operands is translated to a constant in MSFOL.

5.4.4 Change in Step 5 - Return instances

Portus uses the interpretation of the constant instead of the membership predicate to determine the elements belonging to a signature.

5.4.5 Evaluation

Table 5.4 demonstrates the effect of using constants instead of predicates on the performance of Portus. The models `filesystem`, `lights` and `sudoku` have one, three and nine signatures that are translated to constants respectively. The `sudoku` models are the representation of a Sudoku puzzle in Alloy where the hints are encoded using additional constraints. The template for the Sudoku puzzles in Alloy is taken from [61]. For the evaluation of our optimization, we drew Sudoku problems of varying difficulty from [2]. Each of the `sudoku` models has a `run` command, a scope of exactly 9 and is satisfiable

	filesystem			lights			sudoku		
Scope	14	20	26	50	75	100	easy	med	hard
w/o opt	3.21	163.24	t/o	2.68	7.10	14.33	11.28	12.48	10.38
w opt	1.93	139.04	471.44	1.54	5.10	14.33	5.07	5.34	5.62

(a) Performance Results (in seconds)
(t/o = timed out after 30 minutes)

	filesystem		lights		sudoku	
	Scope = 20		Scope = 75		Diff = med	
	w/o opt	w opt	w/o opt	w opt	w/o opt	w opt
Predicates	5	4	9	6	14	5
Constants	0	1	0	3	0	9
Quant Depth	21	20	6	6	5	5
Skolem Constants	2	1	8	5	9	0
Skolem Functions	191	190	6	3	65	1
Initial Term Count	×1	×0.99	×1	×0.81	×1	×0.59
Final Term Count	×1	×0.99	×1	×0.96	×1	×0.99
Transform Time (s)	0.13	0.14	0.55	0.54	1.72	1.61

(b) Profiling Characteristics

Table 5.4: Optimization of signatures using constants in Portus

with a unique solution. All other models have a `check` command, non-exact scopes and are unsatisfiable.

The results in Table 5.4a demonstrate that using constants instead of predicates improves the performance of Portus. As shown in Table 5.4b, even replacing one predicate with a constant has a significant effect on performance especially for larger scopes, as in the case of the benhcmark `filesystem`. The total term count before and after translation to EUFBV shows a clearly positive effect on applying the optimization. One notable result is the effect of constants on the number of skolem constants and functions created during skolemization by Fortress, particulary for the `sudoku` models.

5.5 Optimization of Relations

As mentioned in Section 4.2, relations are declared as fields of signatures in Alloy which are translated to predicates in Portus. These field declarations can contain multiplicity keywords in them. If the declaration has a range of multiplicity `one`, such as:

```
sig A { f: one B }
```

or:

```
sig A { f: e -> one C }
```

where `A`, `B` and `C` are signatures and `e` is an Alloy expression, we can represent `f` as a function instead of a predicate in Portus. Since functions in MSFOL are total, all elements in the domain of the relation are part of the instance. As a result, this optimization can only be applied when all the signatures in the domain of `f` have scopes of an exact size and are top-level signatures. This condition ensures that all elements in the domain of `f` are also part of the instance which is not possible for subsignatures and signatures with non-exact scopes.

5.5.1 Change in Step 1 - Translate signatures

The translation of the relation declaration `f: e -> one B`:

$$f : sort_1(\texttt{e}) \times \cdots \times sort_n(\texttt{e}) \times sort_1(\texttt{B}) \rightarrow Bool$$

is modified to a function declaration in MSFOL:

$$f : sort_1(\texttt{e}) \times \cdots \times sort_n(\texttt{e}) \rightarrow sort_1(\texttt{B})$$

where $arity(\texttt{e}) = n$. The Alloy formula representing the range multiplicity of `f` is excluded from the translation.

5.5.2 Change in Step 3 - Translate formulas

The translation of the set membership expression:

$$[\![(x_1, \ldots, x_n) \in \texttt{f}]\!] \coloneqq f(x_1, \ldots, x_n)$$

is modified to:

$$[\![(x_1, \ldots, x_n) \in \texttt{f}]\!] \coloneqq f(x_1, \ldots, x_{n-1}) = x_n$$

where `f` is translated to a function f in MSFOL.

The translation of the subset predicate:

$$[\![\texttt{e1 in e2}]\!] \coloneqq \forall x_1 : sort_1(\texttt{e1}), \ldots, x_n : sort_n(\texttt{e1}) \bullet [\![(x_1, \ldots, x_n) \in \texttt{e1}]\!] \Rightarrow [\![(x_1, \ldots, x_n) \in \texttt{e2}]\!]$$

is modified to:

$$[\![\texttt{e1 in e2}]\!] \coloneqq \forall x_1 : sort_1(\texttt{e1}), \ldots, x_{n-1} : sort_{n-1}(\texttt{e1}) \bullet [\![(x_1, \ldots, x_{n-1}, f(x_1, \ldots, x_{n-1})) \in \texttt{e2}]\!]$$

if $[\![\texttt{e1}]\!]$ is of the form $f(x_1, \ldots, x_{n-1}) = x_n$, thus, removing the universally quantified variable x_n. There is no simplification to be done for `e1 in e2` if $[\![\texttt{e2}]\!]$ is of the form $f(x_1, \ldots, x_{n-1}) = x_n$. If `e1` and `e2` are both translated to applications of functions f_1 and f_2 respectively, the subset predicate is translated as:

$$[\![\texttt{e1 in e2}]\!] \coloneqq \forall x_1 : sort_1(\texttt{e1}), \ldots, x_{n-1} : sort_{n-1}(\texttt{e1}) \bullet f_1(x_1, \ldots, x_{n-1}) = f_2(x_1, \ldots, x_{n-1})$$

and the equals predicate as:

$$[\![\texttt{e1 = e2}]\!] \coloneqq \forall x_1 : sort_1(\texttt{e1}), \ldots, x_{n-1} : sort_{n-1}(\texttt{e1}) \bullet f_1(x_1, \ldots, x_{n-1}) = f_2(x_1, \ldots, x_{n-1})$$

Note that the formula for subset and equals operators is identical if `e1` and `e2` are both translated to applications of functions. If $[\![\texttt{e1}]\!]$ represents the application of function f, the translation for the join operator:

$$[\![(x_1, ..., x_n) \in \texttt{e1 . e2}]\!] \coloneqq \exists y : sort_1(\texttt{e2}) \bullet [\![(x_1, \ldots, x_m, y) \in \texttt{e1}]\!] \wedge [\![(y, x_{m+1}, \ldots, x_n) \in \texttt{e2}]\!]$$

is modified to:

$$[\![(x_1, \ldots, x_n) \in \texttt{(e1 . e2)}]\!] \coloneqq [\![(f(x_1, \ldots, x_m), x_{m+1}, \ldots, x_n) \in \texttt{e2}]\!]$$

where $m < n$, $arity(\texttt{e1}) = m + 1$ and $arity(\texttt{e2}) = n - m + 1$.

5.5.3 Change in Step 5 - Return instances

PORTUS uses the interpretation of the function to determine the tuples in the relation. A tuple is formed by combining the arguments of the function with the output value.

	ceilingsAndFloors			lights			lists		
Scope	10	15	20	60	90	120	15	20	25
w/o opt	0.83	9.58	79.75	4.62	19.43	33.45	1.30	17.00	78.99
w opt	0.03	0.04	0.06	1.20	2.90	7.38	0.10	0.24	0.45

(a) Performance Results (in seconds)

	ceilingsAndFloors		lights		lists	
	Scope = 15		Scope = 90		Scope = 20	
	w/o opt	w opt	w/o opt	w opt	w/o opt	w opt
Functions	0	2	0	1	0	2
Predicates	4	2	9	8	6	4
Max Arity	2	1	3	2	2	1
Quant Depth	3	2	6	6	20	20
Skolem Functions	6	1	6	2	3	1
Initial Term Count	×1	×0.61	×1	×0.87	×1	×0.95
Final Term Count	×1	×0.21	×1	×0.42	×1	×0.46
Transform Term (s)	0.05	0.02	0.89	0.45	0.26	0.17

(b) Profiling Characteristics

Table 5.5: Optimization of relations using functions in Portus

5.5.4 Evaluation

Table 5.5 demonstrates the effect of using functions instead of predicates on the performance of Fortress. The models `lights`, `ceilingsAndFloors` and `lists` have one, two and two relations that are translated to functions respectively and the `lists` model contains a set hierarchy. All models are assigned exact scopes as is required by the optimization criteria.

Based on the results in Table 5.5a, functions have a very significant impact on solving time especially on greater scopes. Another observation is the total term count before and after translation to EUFBV decreases by a significant factor after applying the optimization in Table 5.5b.

```
A in first.*next
no next.first
all v2 : A | v2 = first or one next.v2
all v2 : A | v2 = A - next.A or one v2.next
all v2 : A | not (v2 in v2.^next)
```

(a) Facts

```
first() : Ord.First          max(e) : e - e.^(~(Ord.Next))
next() : Ord.Next            min(e) : e - e.^(Ord.Next)
last() : A - next.A          lt(e1,e2) : e1 in prevs.e2
prev() : ~(Ord.Next)         gt(e1,e2) : e1 in nexts.e2
prevs(e) : e.^(~(Ord.Next)) lte(e1,e2) : e1 = e2 or lt(e1,e2)
nexts(e) : e.^(Ord.Next)     gte(e1,e2) : e1 = e2 or gt(e1,e2)
```

(b) Relations

Figure 5.1: Ordering Module in Alloy

5.6 Optimization of Ordering Module

The ordering module is used to impose a linear ordering on the elements of a signature. It also implicitly sets the scope of that signature to be exact. The ordering module on signature `A` includes the following declarations:

```
1 sig Ord {
2     First: set A,
3     Next: A -> A
4 }
```

with the facts mentioned in Figure 5.1a. Alloy also provides several relations for the ordering module as listed in Figure 5.1b.

Since the elements of a set are interchangeable in Alloy, we can apply symmetry breaking restrictions to select one ordering of the domain elements. Calls to any of the relations mentioned above by the model can be appropriately replaced by an MSFOL formula or an application of an MSFOL predicate.

$$
\begin{array}{rcl}
[\![x \in \texttt{first}]\!] & \coloneqq & x = 1_A \\
[\![(x_1, x_2) \in \texttt{next}]\!] & \coloneqq & next(x_1, x_2) \\
[\![x \in \texttt{last}]\!] & \coloneqq & x = k_A \\
[\![(x_1, x_2) \in \texttt{prev}]\!] & \coloneqq & prev(x_1, x_2) \\
[\![x \in \texttt{prevs[e]}]\!] & \coloneqq & \exists\, v : A \bullet [\![v \in \texttt{e}]\!] \wedge prevs(x, v) \\
[\![x \in \texttt{nexts[e]}]\!] & \coloneqq & \exists\, v : A \bullet [\![v \in \texttt{e}]\!] \wedge nexts(x, v) \\
[\![x \in \texttt{max[e]}]\!] & \coloneqq & [\![x \in \texttt{e}]\!] \wedge \neg\, [\![x \in \texttt{prevs[e]}]\!] \\
[\![x \in \texttt{min[e]}]\!] & \coloneqq & [\![x \in \texttt{e}]\!] \wedge \neg\, [\![x \in \texttt{nexts[e]}]\!] \\
[\![\texttt{lt[e1,e2]}]\!] & \coloneqq & \exists\, v_1, v_2 : A \bullet [\![v_1 \in \texttt{e1}]\!] \wedge [\![v_2 \in \texttt{e2}]\!] \wedge prevs(v_1, v_2) \\
[\![\texttt{gt[e1,e2]}]\!] & \coloneqq & \exists\, v_1, v_2 : A \bullet [\![v_1 \in \texttt{e1}]\!] \wedge [\![v_2 \in \texttt{e2}]\!] \wedge nexts(v_1, v_2) \\
[\![\texttt{lte[e1,e2]}]\!] & \coloneqq & [\![\text{e1} = \text{e2}]\!] \vee [\![\texttt{lt[e1,e2]}]\!] \\
[\![\texttt{gte[e1,e2]}]\!] & \coloneqq & [\![\text{e1} = \text{e2}]\!] \vee [\![\texttt{gt[e1,e2]}]\!]
\end{array}
$$

Figure 5.2: Optimization of ordering module in PORTUS for sig `A` with scope k

5.6.1 Change in Step 1 - Translate signatures

The signature `Ord` and the fields `First` and `Next` are ignored during the first step of translation. No sorts or predicate symbols are introduced during this stage.

5.6.2 Change in Step 3 - Translate formulas

Figure 5.2 illustrates the translation for all relations provided by the ordering module for signature `A` with scope k.

The ordering on domain elements can be used to apply symmetry breaking optimizations on the binary MSFOL predicates, *next*, *prev*, *nexts* and *prevs*, of the form $A \times A \to Bool$ introduced by the translation of ordering module functions. Unfortunately, we cannot replace the relations *next* and *prev* with functions instead of predicates in MSFOL because they are not total. The symmetry breaking formulas for a set whose elements are required to be in a linear order force only the following interpretations to be considered:

$$
\begin{array}{rcl}
next & \coloneqq & \{(1_A, 2_A), (2_A, 3_A), \ldots, (k - 1_A, k_A)\} \\
prev & \coloneqq & \{(2_A, 1_A), (3_A, 2_A), \ldots, (k_A, k - 1_A)\} \\
nexts & \coloneqq & \{(1_A, 2_A), \ldots, (1_A, k_A), (2_A, 3_A), \ldots, (2_A, k_A), \ldots, (k - 1_A, k_A)\} \\
prevs & \coloneqq & \{(2_A, 1_A), (3_A, 2_A), (3_A, 1_A), \ldots, (k_A, k - 1_A), \ldots, (k_A, 1_A)\}
\end{array}
$$

The symmetry breaking formulas limit the interpretations for the predicates more strongly

than what is done by Fortress's symmetry breaking. The following formulas are added to the MSFMF problem to restrict the interpretation of *next*:

$$
\begin{array}{cccc}
\neg\, next(1_A, 1_A) & \neg\, next(2_A, 1_A) & \ldots & \neg\, next(k-1_A, 1_A) \quad \neg\, next(k_A, 1_A) \\
next(1_A, 2_A) & \neg\, next(2_A, 2_A) & \ldots & \neg\, next(k-1_A, 2_A) \quad \neg\, next(k_A, 2_A) \\
\neg\, next(1_A, 3_A) & next(2_A, 3_A) & \ldots & \neg\, next(k-1_A, 3_A) \quad \neg\, next(k_A, 3_A) \\
\ldots & \ldots & \ldots & \ldots \quad \ldots \\
\neg\, next(1_A, k_A) & \neg\, next(2_A, k_A) & \ldots & next(k-1_A, k_A) \quad \neg\, next(k_A, k_A)
\end{array}
$$

Similar formulas are added for the other predicates based on their interpretations. Note that these predicates and formulas are only introduced when they are required.

5.6.3 Evaluation

The ordering module is the most widely used built-in module in Alloy models. Table 5.6 demonstrates the effect of using symmetry breaking for the translation of the ordering module. The models `addressBook3d`, `hotel2` and `ringElection1` have one, two and two signatures with ordering respectively. These models contain calls to the following Alloy constants and functions:

- `addressBook3d` : `first`, `last`, `next`
- `hotel2` : `last`, `next`, `nexts`
- `ringElection1` : `first`, `last`, `next`, `prev`, `prevs`

resulting in the following auxiliary predicates introduced by Portus during the translation:

- `addressBook3d` : *next*
- `hotel2` : *next*, *nexts*
- `ringElection1` : *next*, *prev*, *prevs*

For each model, all signatures except the ones with ordering have non-exact scopes.

The results in Table 5.6b show that even if total terms in the set of MSFOL formulas in the generated MSFMF problem is more than the unoptimized MSFMF problem, as in model `ringElection1`, the ordering module optimization has a significant impact on the performance of Portus as shown in Table 5.6a and on the number of skolem functions because of the join operators avoided.

	addressBook3d			hotel2			ringElection1		
Scope	5	6	7	3	5	7	6	8	10
w/o opt	0.63	5.86	67.02	4.01	35.94	99.63	2.78	10.78	68.09
w opt	0.35	3.44	65.34	1.78	11.04	57.83	1.48	3.30	7.86

(a) Performance Results (in seconds)

	addressBook3d		hotel2		ringElection1	
	Scope = 6		Scope = 5		Scope = 8	
	w/o opt	w opt	w/o opt	w opt	w/o opt	w opt
Sorts	3	2	7	5	4	2
Predicates	11	9	16	12	12	9
Depth Quant	12	12	10	8	8	7
Skolem Functions	33	24	312	115	65	26
Initial Term Count	×1	×0.98	×1	×0.58	×1	×1.16
Final Term Count	×1	×0.99	×1	×0.35	×1	×0.45
Transform Time (s)	1.17	1.23	2.15	0.82	0.13	0.07

(b) Profiling Characteristics

Table 5.6: Optimization of ordering module in Portus

5.7 Optimization of Transitive Closure

The basic approach to translate transitive closure and reflexive transitive closure for finite domains is mentioned in Section 4.4.5. In the following subsections, we discuss four more approaches in detail and compare the performance of all techniques on increasing scopes.

5.7.1 Change in Step 3 - Translate Formulas

Unlike the basic approach, the approaches discussed below introduce auxiliary predicate or function symbols along with a set of MSFOL formulas.

Approach 1 - Iterative Squaring

The iterative squaring technique introduced by Burch et al. [14] translates the transitive closure of binary relation `R` of type `A -> A`, where `A` is an Alloy signature with scope k, into $\lceil log(k) \rceil$ formulas.

$\texttt{R}_0 = \texttt{R}$

$\texttt{R}_{i+1} = \texttt{R}_i \texttt{ . } \texttt{R}_i \texttt{ + } \texttt{R}_i$

Given an Alloy expression `e` of type `A -> A`, where `A` is an Alloy signature with scope k, Portus introduces:

- auxiliary binary predicates $R_1, R_2, \ldots, R_c : A \times A \rightarrow Bool$, where $c = \lceil log(k) \rceil - 1$.

To limit these auxiliary predicates, Portus adds the following formulas:

$$\begin{aligned}
&\forall x, y : A \bullet R_1(x, y) \Leftrightarrow [\![(x, y) \in \texttt{e}]\!] \vee \exists z : A \bullet [\![(x, z) \in \texttt{e}]\!] \wedge [\![(z, y) \in \texttt{e}]\!] \\
&\forall x, y : A \bullet R_2(x, y) \Leftrightarrow R_1(x, y) \vee \exists z : A \bullet R_1(x, z) \wedge R_1(z, y) \\
&\quad \ldots \\
&\forall x, y : A \bullet R_c(x, y) \Leftrightarrow R_{c-1}(x, y) \vee \exists z : A \bullet R_{c-1}(x, z) \wedge R_{c-1}(z, y)
\end{aligned}$$

and translates the closure operators as:

$$\begin{aligned}
[\![(x, y) \in \texttt{\^{}e}]\!] &\coloneqq R_c(x, y) \vee \exists z : A \bullet R_c(x, z) \wedge R_c(z, y) \\
[\![(x, y) \in \texttt{*e}]\!] &\coloneqq x = y \vee [\![(x, y) \in \texttt{\^{}e}]\!]
\end{aligned}$$

Approach 2 - Claessen's Axiomatisation

The axiomatisation of reflexive transitive closure in first-order logic over finite domains was first introduced by Claessen [15]. Given an Alloy expression `e` of type `A -> A`, where `A` is an Alloy signature with scope k, Claessen introduces:

- an auxiliary binary function $s : A \times A \rightarrow A$ where $s(x, y) = z$ indicates that z is a next step in the path from x towards y,
- an auxiliary ternary predicate $C : A \times A \times A \rightarrow Bool$ where $C(x, y, z)$ indicates that y is closer to z than x,

– a binary predicate $^*R : A \times A \to Bool$ representing the reflexive transitive closure of `e`,

– and optionally, a binary predicate $\hat{}R : A \times A \to Bool$ representing the transitive closure of `e`.

The function and predicate symbols are limited by the following formulas:

$$\begin{aligned}
&\forall x, y, z, u : A \bullet C(x, y, u) \wedge C(y, z, u) \Rightarrow C(x, z, u) \\
&\forall x, y : A \bullet \neg\, C(x, x, y) \\
&\forall x, y : A \bullet {}^*R(x, y) \wedge \neg(x = y) \Rightarrow C(x, s(x, y), y) \\
&\forall x, y : A \bullet {}^*R(x, y) \wedge \neg(x = y) \Rightarrow {}^*R(s(x, y), y) \\
&\forall x, y : A \bullet {}^*R(x, y) \wedge \neg(x = y) \Rightarrow [\![(x, s(x, y)) \in \mathtt{e}]\!] \\
&\forall x, y, z : A \bullet {}^*R(x, y) \wedge {}^*R(y, z) \Rightarrow {}^*R(x, z) \\
&\forall x, y : A \bullet [\![(x, y) \in \mathtt{e}]\!] \Rightarrow {}^*R(x, y) \\
&\forall x : A \bullet {}^*R(x, x)
\end{aligned}$$

The transitive closure of expression `e` is defined using the formula:

$$\forall x, y : A \bullet \hat{}R(x, y) \Leftrightarrow \exists z : A \bullet [\![(x, z) \in \mathtt{e}]\!] \wedge {}^*R(z, y)$$

Using this approach, PORTUS introduces function s and predicates C and $^*R$, adds the formulas above limiting the auxiliary symbols and translates the closure operators as:

$$\begin{aligned}
[\![(x, y) \in \mathtt{\hat{}e}]\!] &:= \exists z : A \bullet [\![(x, z) \in \mathtt{e}]\!] \wedge {}^*R(z, y) \\
[\![(x, y) \in \mathtt{*e}]\!] &:= {}^*R(x, y)
\end{aligned}$$

Approach 3 - Eijck's Axiomatisation

Inspired by Claessen's work [15], an alternative axiomatisation of reflexive transitive closure in first-order logic over finite domains was introduced by Eijck [66]. Given an Alloy expression `e` of type `A -> A`, where `A` is an Alloy signature with scope k, Eijck introduces:

– an auxiliary ternary predicate $C : A \times A \times A \to Bool$ where $C(x, y, z)$ indicates that y is at some finite non-zero distance from the shortest path from x to z,

– a binary predicate $^*R : A \times A \rightarrow Bool$ representing the reflexive transitive closure of `e`,

– and optionally, a binary predicate $\hat{}R : A \times A \rightarrow Bool$ representing the transitive closure of `e`.

The auxiliary predicate C is limited by the following formulas:

$$\forall x, y, z, u : A \bullet C(x, y, u) \land C(y, z, u) \Rightarrow C(x, z, u)$$
$$\forall x, y : A \bullet \neg\, C(x, x, y)$$
$$\forall x, y, z : A \bullet C(x, y, y) \land C(y, z, z) \land \neg(x = z) \Rightarrow C(x, z, z)$$
$$\forall x, y, z : A \bullet C(x, y, z) \land \neg(y = z) \Rightarrow C(y, z, z)$$
$$\forall x, y : A \bullet [\![(x, y) \in \mathtt{e}]\!] \land \neg(x = y) \Rightarrow C(x, y, y)$$
$$\forall x, y : A \bullet C(x, y, y) \Rightarrow \exists z : A \bullet [\![(x, z) \in \mathtt{e}]\!] \land C(x, z, y)$$

The reflexive transitive closure of expression `e` is defined using the formula:

$$\forall x, y : A \bullet {}^*R(x, y) \Leftrightarrow C(x, y, y) \lor x = y$$

Similar to Claessen's approach, the transitive closure of `e` is defined using the formula:

$$\forall x, y : A \bullet \hat{}R(x, y) \Leftrightarrow \exists z : A \bullet [\![(x, z) \in \mathtt{e}]\!] \land {}^*R(z, y)$$

Unlike Claessen's approach, Eijck's approach separates the definition of reflexive transitive closure from the formulas defining the auxiliary predicates. This fact combined with the top-down approach introduced by Portus allows us to remove the predicates for the closure operators (along with their formulas). Portus introduces predicate C, adds the formulas above limiting C and translates the closure operators as:

$$\begin{array}{lcl} [\![(x, y) \in \mathtt{\hat{}e}]\!] & \coloneqq & \exists z : A \bullet [\![(x, z) \in \mathtt{e}]\!] \land (C(z, y, y) \lor z = y) \\ [\![(x, y) \in \mathtt{*e}]\!] & \coloneqq & C(x, y, y) \lor x = y \end{array}$$

Approach 4 - Liu et al.'s Axiomatisation

Inspired by Claessen's work [15], another axiomatisation of reflexive transitive closure in first-order logic over finite domains using integers was introduced by Liu et al [39]. Given an Alloy expression `e` of type `A -> A`, where `A` is an Alloy signature with scope k, Liu et al. introduces:

– an auxiliary binary function $P : A \times A \to Int$ where $P(x, y)$ represents the smallest number of steps as an integer value required to reach y from x.

The auxiliary function P is limited by the following formulas:

$$\forall x, y : A \bullet [\![(x, y) \in \texttt{e}]\!] \Leftrightarrow P(x, y) = 1$$
$$\forall x, y, z : A \bullet P(x, y) > 0 \land P(y, z) > 0 \Rightarrow P(x, z) > 0$$
$$\forall x, y : A \bullet P(x, y) > 1 \Rightarrow \exists z : A \bullet P(x, z) = 1 \land P(x, y) = P(z, y) + 1$$

The transitive closure of expression `e` is defined using the formula:

$$\forall x, y : A \bullet \hat{}R(x, y) \Leftrightarrow P(x, y) > 0$$

Using this approach, Portus introduces function P, adds the formulas above limiting P and translates the closure operators as:

$$\begin{aligned} [\![(x, y) \in \texttt{\^{}e}]\!] &:= P(x, y) > 0 \\ [\![(x, y) \in \texttt{*e}]\!] &:= P(x, y) > 0 \lor x = y \end{aligned}$$

Since integers in Portus are bounded by a scope or bit-width, the scope for integers must be large enough to include the maximum value of P i.e. the scope of `A`.

5.7.2 Evaluation

Table 5.7 compares the closure approaches mentioned above with the basic approach discussed in Section 4.4.5 and with each other. The models `filesystem`, `addressBook2e` and `grandpa` use one of the closure operators on a binary relation, join of a variable and ternary relation and an Alloy expression of the form `e1+e2` with arity 2 respectively. In the interest of space, we only show the profiling characteristics for one of the models in Table 5.7b. The profiling characteristics for the other two models are included in Appendix C. None of the models used transitive closure on integers (which are allowed to be unbounded as described in Section 5.9).

The performance results for each approach on increasing scopes are shown in Table 5.7a. Although the term count of the basic approach after translation to EUFBV is the smallest, as shown in Table 5.7b, it is not the best approach in performance as compared to the others. The basic approach performs considerably well on the model `filesystem` but

	filesystem			addressBook2e			grandpa		
Scope	9	12	15	9	7	11	16	20	24
basic	0.23	3.57	10.52	63.53	oom	oom	oom	oom	oom
Burch et al.	1.91	7.67	37.13	1.75	2.50	206.64	1.04	1.76	951.43
Claessen	0.33	23.51	t/o	0.70	2.42	10.29	2.10	9.67	15.66
Eijck	0.29	1.73	5.89	0.78	2.86	15.59	2.17	17.12	27.42
Liu et al.	3.24	t/o	t/o	0.35	0.97	1.70	0.58	1.52	2.52

(a) Performance Results (in seconds)
(t/o = timed out after 30 minutes, oom = out of memory)

	filesystem				
	Scope = 12				
	basic	Burch et al.	Claessen	Eijck	Liu et al.
Functions	0	0	1	0	1
Predicates	5	9	7	6	5
Max Arity	2	2	3	3	2
Depth Quant	13	3	4	4	3
Skolem Constants	2	2	2	2	2
Skolem Functions	67	6	1	2	2
Initial Term Count	×1	×0.34	×0.36	×0.36	×0.29
Final Term Count	×1	×2.53	×7.93	×8.3	×1.09
Transform Time (s)	0.28	0.26	0.36	0.41	0.23

(b) Profiling Characteristics

Table 5.7: Comparison of closure translations in Portus

the solver runs out of memory on the other two models. The approach by Burch et al. gives great performance on smaller scopes but does not scale well as the scope increases. Although the approach by Claessen performs slightly better than the similar approach by Eijck on two of the models, Claessen's approach times out on the `filesystem` model. The approach by Liu et al. performs the best on two of the models but times out for the

`filesystem` model. Based on the results, we choose the approach by Eijck to translate the closure operators since it performs well on all three models.

5.8 Optimization of Cardinality

Any expressions using the cardinality operator to compare with an integer constant in Alloy can be axiomatised in first-order logic without the use of integers.

5.8.1 Change in Step 3 - Translate Formulas

The translation for all integer comparison operators in Alloy is given in Figure 5.3. A syntactic ordering is applied on the expressions containing comparison operators with cardinality where `#e = n` is equivalent to `n = #e`, `#e >= n` is equivalent to `n <= #e`, and so on.

5.8.2 Evaluation

Table 5.8 presents the results of applying the optimization on set cardinality compared with an integer constant. The models `addressBook1c`, `filesystem` and `mercurial` use the comparison operators `>1`, `=6` and `<=2` on relations of arity three, one and two respectively. The scope of integers for the unoptimized translation is set appropriately to include the maximum value for cardinality.

Although the optimization may result in a higher term count, as shown in Table 5.8b for `filesystem` before and after translation to EUFBV and for `mercurial` after translation to EUFBV, the results in Table 5.8a show that removing the function introduced to represent cardinality of a relation and using quantifiers to express the cardinality constraint has a significant impact on the performance of Portus.

5.9 Optimization of Integers

As mentioned in Section 2.3, Fortress provides two options for integers: modular arithmetic and unbounded. The modular arithmetic option is included in the basic translation in Chapter 4. The modular arithmetic option can produce spurious counterexamples. Since

$$
\begin{aligned}
[\![\texttt{\#e = c}]\!] \;\coloneqq\; & \exists x_{1:1}, \ldots, x_{1:c} : sort_1(\texttt{e}) \bullet \ldots \bullet \exists x_{n:1}, \ldots, x_{n:c} : sort_n(\texttt{e}) \bullet \\
& \quad \forall x_{1:c+1} : sort_1(\texttt{e}), \ldots, x_{n:c+1} : sort_n(\texttt{e}) \bullet \\
& \quad\quad (\neg(x_{1:1} = x_{1:2}) \vee \ldots \vee \neg(x_{n:1} = x_{n:2})) \wedge \\
& \quad\quad\quad \ldots \\
& \quad\quad (\neg(x_{1:1} = x_{1:c}) \vee \ldots \vee \neg(x_{n:1} = x_{n:c})) \wedge \\
& \quad\quad (\neg(x_{1:2} = x_{1:3}) \vee \ldots \vee \neg(x_{n:2} = x_{n:3})) \wedge \\
& \quad\quad\quad \ldots \\
& \quad\quad (\neg(x_{1:2} = x_{1:c}) \vee \ldots \vee \neg(x_{n:2} = x_{n:c})) \wedge \\
& \quad\quad\quad \ldots \\
& \quad\quad (\neg(x_{1:c-1} = x_{1:c}) \vee \ldots \vee \neg(x_{n:c-1} = x_{n:c})) \wedge \\
& \quad\quad ([\![(x_{1:c+1}, \ldots, x_{n:c+1}) \in \texttt{e}]\!] \Leftrightarrow \\
& \quad\quad\quad (x_{1:1} = x_{1:c+1} \wedge \ldots \wedge x_{n:1} = x_{n:c+1}) \vee \\
& \quad\quad\quad\quad \ldots \\
& \quad\quad\quad (x_{1:c} = x_{1:c+1} \wedge \ldots \wedge x_{n:c} = x_{n:c+1})) \\
[\![\texttt{\#e <= c}]\!] \;\coloneqq\; & \forall x_{1:1}, \ldots, x_{1:c+1} : sort_1(\texttt{e}) \bullet \ldots \bullet \forall x_{n:1}, \ldots, x_{n:c+1} : sort_n(\texttt{e}) \bullet \\
& \quad\quad [\![(x_{1:1}, \ldots, x_{n:1}) \in \texttt{e}]\!] \wedge \ldots \wedge [\![(x_{1:c+1}, \ldots, x_{n:c+1}) \in \texttt{e}]\!] \Rightarrow \\
& \quad\quad\quad (x_{1:1} = x_{1:2} \wedge \ldots \wedge x_{n:1} = x_{n:2}) \vee \\
& \quad\quad\quad\quad \ldots \\
& \quad\quad\quad (x_{1:1} = x_{1:c+1} \wedge \ldots \wedge x_{n:1} = x_{n:c+1}) \vee \\
& \quad\quad\quad (x_{1:2} = x_{1:3} \wedge \ldots \wedge x_{n:2} = x_{n:3}) \vee \\
& \quad\quad\quad\quad \ldots \\
& \quad\quad\quad (x_{1:1} = x_{2:c+1} \wedge \ldots \wedge x_{n:2} = x_{n:c+1}) \vee \\
& \quad\quad\quad\quad \ldots \\
& \quad\quad\quad (x_{1:c} = x_{1:c+1} \wedge \ldots \wedge x_{n:c} = x_{n:c+1}) \\
[\![\texttt{\#e < c}]\!] \;\coloneqq\; & [\![\texttt{\#e <= c-1}]\!] \\
[\![\texttt{\#e >= c}]\!] \;\coloneqq\; & \neg\, [\![\texttt{\#e < c}]\!] \\
[\![\texttt{\#e > c}]\!] \;\coloneqq\; & [\![\texttt{\#e >= c+1}]\!]
\end{aligned}
$$

Figure 5.3: Optimization of cardinality in Portus

SMT solvers support integers as one of their built-in theories, we investigated the other option by employing a hybrid approach where all uninterpreted sorts are bounded by scopes but the integers are left unbounded.

Portus creates an MSFOL problem using unbounded integers, which is not necessarily decidable. If the integer expressions in the resulting MSFOL problem use only linear arithmetic, the problem is decidable since the combination of EUF and linear arithmetic is decidable using the Nelson-Oppen method [35, 43]. The Z3 SMT solver uses the Nelson-

	addressBook1c			filesystem			mercurial		
Scope	20	25	30	9	12	15	9	12	15
w/o opt	8.42	14.62	61.22	4.58	21.78	226.20	1.94	14.56	63.10
w opt	1.65	4.18	8.21	2.65	15.50	77.30	2.01	10.76	54.23

(a) Performance Results (in seconds)

	addressBook1c		filesystem		mercurial	
	Scope = 25		Scope = 12		Scope = 12	
	w/o opt	w opt	w/o opt	w opt	w/o opt	w opt
Functions	1	0	1	0	1	0
Depth Quant	4	5	3	7	4	4
Skolem Constants	2	8	2	8	4	4
Initial Term Count	×1	×0.02	×1	×1.15	×1	×0.79
Final Term Count	×1	×0.85	×1	×1.01	×1	×2.59
Transform Time (s)	1.04	0.98	0.26	0.26	0.12	0.24

(b) Profiling Characteristics

Table 5.8: Optimization of cardinality in Portus

Oppen method to incrementally solve a combination of theories [19].

As discussed in Section 4.4.7, Portus defines cardinality over finite relations using integers. Since we use only equality and addition in our translation of the cardinality operator, the basic translation of set cardinality stays within Presburger arithmetic theory [47], a subset of linear arithmetic theory, which is decidable in combination with EUF.

5.9.1 Change in Step 2 - Translate Scopes

Portus does not specify a scope for integers in the MSFOL problem.

5.9.2 Change in Step 4 - Solve using Fortress

The unbounded option for integers is chosen in Fortress.

	magicSquare			account			handshake		
Scope	3	4	5	5	5	5	12	14	16
Bitwidth	5	7	8	20	30	40	5	7	7
bounded	2.41	32.63	t/o	1.12	32.61	62.80	31.35	68.12	534.92
unbounded	3.42	19.35	469.32	0.76	2.34	3.69	31.05	67.60	482.79

Table 5.9: Performance Results (in seconds) for optimization of integers in Portus (t/o = timed out after 30 minutes)

5.9.3 Evaluation

Table 5.9 demonstrates the results for leaving integers unbounded compared to the modular arithmetic option for bounded integers in Fortress. The models `magicSquare` and `account` contain integers in relation declarations and the `handshake` model indirectly uses integers via the cardinality operator. These models contain formulas that include the following Alloy integer operators:

- `magicSquare` : `+`, `=`
- `account` : `+`, `-`, `=`, `<=`
- `handshake` : `=`

Since Fortress only replaces the integer sorts and operators with bit-vector sorts and operators, there are no significant differences in any model characteristics and the profiling results are not shown.

The results show that although the difference in the performance of Portus is not that noticeable for the model containing cardinality, the performance for unbounded integers is improved significantly as compared to bounded integers on the models using integers directly, particularly noticeable for the `magicSquare` model where the solver times out for bounded integers.

5.10 Conclusion

This chapter discussed the possible optimizations for some common cases in Alloy. The optimized translation is discussed in detail and compared to the basic translation provided in Chapter 4. For the final evaluation of our library Portus against Kodkod, we choose

to apply all of the optimizations, use the approach by Eijck [66] to translate the closure operators and leave integers unbounded.

Chapter 6

Experimental Results

In this chapter, we evaluate the performance of our library Portus integrated with the Alloy Analyzer, with the optimizations discussed in Chapter 5, compared to the Alloy Analyzer using Kodkod. We determine the correctness of Portus by extensive testing on Alloy models. We start by briefly discussing the implementation details and the experimental setup for our performance and correctness testing.

6.1 Implementation

Portus is implemented in Java as an extension of the main Alloy code base[1]. As a result, it is fully integrated with the Alloy Analyzer. We use the Fortress library implemented by Poremba et al. [46] in Scala to represent our problem in MSFOL. Portus includes Fortress as a submodule so that any upgrades to the Fortress library in the future can be easily updated in the Alloy tool. We utilize a cache in our implementation to identify similar expressions so any auxiliary functions created by our translation can be re-used.

6.2 Experimental Setup

We use a collection of 190 Alloy models scraped from public Github repositories[2] to evaluate the performance and correctness of our library Portus. All experiments were run on

[1] Available at https://github.com/WatForm/portus.

[2] These models were collected by Amin Bandali.

Intel®Xeon®CPU E3-1240 v5 @ 3.50 GHz with Ubuntu 16.04 64-bit with up to 64GB of user memory. The basic translation in Chapter 4 is combined with all the optimizations discussed in Chapter 5 and Eijck's closure axiomatisation for the evaluation of Portus. We use the `v2si` model finder with the Z3 SMT solver [20] and unbounded integers option in Fortress. We use the MiniSat solver [56] in Kodkod, same as what was used by Torlak et al. in their initial evaluation of Kodkod [62]. Information about the version of each tool we used is available in Appendix A.

Our performance testing is done in two stages. The first stage finds appropriate starting scope(s) for each command of an Alloy model. The scopes are chosen such that at least one tool takes between 30 seconds and 15 minutes. Since the choice of scope for each individual signature in an Alloy model is very complicated, we used the default scopes specified in each command as initial scopes. We started by doubling the default overall scope and the individual scope for each signature except ones with implicit scopes until at least one tool takes between the specified time range to solve the model. If both of the tools exceed the specified time range, we switch to using binary search between the latest run and the previous run to find the optimal starting scopes. The second stage runs Portus and Kodkod on each model starting from the scopes chosen in the first stage and incrementally increasing the scopes by an appropriate amount. We repeat each experiment 5 times and report the average with a time limit of 15 minutes on each process. Each timeout is assigned a value of 900 seconds during the total time calculation for each tool.

Our correctness testing uses the starting scopes chosen during our first stage of performance testing for each command of an Alloy model and checks the satisfiability of the models and the correctness of a predetermined number of finite interpretations. Each experiment is run only once and any anomalies are noted.

6.3 Performance Testing

The performance of Portus is tested using the 190 Alloy models scraped from public Github repositories. After the first stage of performance testing where an appropriate scope is chosen for each command of an Alloy model, we run Portus and Kodkod on these scopes five times and record the average. From the 190 Alloy models, 37 models timed out on the default scopes for Portus, three models timed out for Kodkod and eight models used constructs not supported in Portus (bit-shifting operators and higher-order quantifications). Of the remaining 142 models, 85 (60%) were satisfiable and 57 (40%) were unsatisfiable. From these 142 models, Portus performed better than Kodkod on

only 7% of the models where the total time taken by Portus was 132 times more than the total time taken by Kodkod. Of the 85 satisfiable and 57 unsatisfiable models, Portus outperforms Kodkod on 3.5% of the satisfiable and 12.3% of the unsatisfiable models (2% and 5% of all models respectively).

In order to analyze these results, we attempted to classify Alloy models using certain characteristics in order to identify the subset of problems Portus performs better on as compared to Kodkod. The 190 Alloy models were profiled based on the criteria mentioned in Chapter 5 for each optimization. Each model is tagged based on if it contains any signatures with exact scopes, a signature hierarchy or if any signature uses the ordering module. In addition, those models are tagged that contain signatures with a scope of exactly one or relations with a range of multiplicity `one`. The models that use the closure, cardinality and integer operators are tagged as well. A model can have multiple tags at the same time. An analysis of the profiling results indicates that although Kodkod wins over Portus on Alloy models primarily containing relational constraints (join, product, closure and so on), Portus seems to win on the models containing integers or relations with a range of multiplicity `one` (functions).

To test our hypothesis that Portus outperforms Kodkod on models with the criteria mentioned above, we use a smaller collection of 51 frequently used Alloy models. This collection of models includes the models provided with the Alloy Analyzer and models used for evaluation in the research papers related to Alloy [27, 31, 41, 57]. From this collection, we pick the models based on if they primarily contain relational operators, integers or relations with a range of multiplicity `one`. For the category of models with relational operators, the models containing functions and integer and cardinality operators are removed. For the category of models with integers, models containing formulas with a heavy use of relational operators are removed. For the category of models with functions, models containing formulas using complicated operators, such as closure and cardinality, are removed and the scopes for appropriate signatures are set to be exact. We further narrow down the models in these three categories to ensure that the models that are chosen are as diverse as possible.

We choose a small subset of problems containing only relational constraints, which we call P_{alloyr}[3], and evaluate each problem on increasing scopes. All models in P_{alloyr} are unsatisfiable and have non-exact scopes. Unsatisfiable models and non-exact scopes are better for the evaluation of any tool because they are usually harder to analyze since the possible number of instances to check are greater. Further information about each model is mentioned below:

[3]Available at https://github.com/WatForm/portus-tests.

	Command	Scope	PORTUS	Kodkod
ceilingsAndFloors	BelowToo″	11	10.48	28.35
		12	10.99	601.82
filesystem	fileInDir	18	18.79	0.03
		19	t/o	0.34
checkFixedSize	initOk	40	244.63	60.22
	readOk	40	254.20	38.08
	writeOk	48	239.59	252.23
addressBook2e	delUndoesAdd	15	55.55	0.46
	addIdempotent	15	41.66	0.55
ringElection1	atMostOneElected	15	73.46	0.13
		25	t/o	1.28
Best out of 11			3	8
Total Time (s)			2749.35	983.49

Table 6.1: Time taken (in seconds) by PORTUS and Kodkod on P_{alloyr}
(The highlighted cell indicates the best result in each row. t/o = timed out after 900s.)

- The `ceilingsAndFloors` model contains the join operator.
- The `filesystem` model contains a signature hierarchy and the relational operators (`.`, `*`).
- The `checkFixedSize` model contains the relational operators (`-`, `.`, `->`, `++`).
- The `addressBook2e` model contains a signature hierarchy and the relational operators (`+`, `&`, `-`, `->`, `.`, `^`).
- The `ringElection1` model contains two signatures with ordering and the relational operators (`.`, `-`, `^`).

As a whole, the models in P_{alloyr} contain signature hierarchy, the ordering module and most of the relational operators (except `~`, `<:`, `:>`) including both the closure operators.

Table 6.1 demonstrates the time taken (in seconds) by each tool to solve the problems in P_{alloyr}. With the exception of a few models, Kodkod performs significantly better than PORTUS. PORTUS times out for two of the models whereas, Kodkod does not time out for any of the models in P_{alloyr}.

Next, we choose a small subset of problems containing integers including the cardinal-

	Command	Scope	Bitwidth	PORTUS	Kodkod
`handshake`	Puzzle	12	5	31.05	0.07
		14	7	67.60	0.08
`square`	show	2	14	0.13	39.02
		2	15	0.17	152.16
`magicSquare`	solve	4	7	19.35	26.85
		5	8	469.32	t/o
`account`	checkBalance	5	11	0.12	72.83
		5	12	0.14	302.94
Best out of 8				6	2
Total Time (s)				587.88	1493.95

Table 6.2: Time taken (in seconds) by PORTUS and Kodkod on P_{alloyi}
(The highlighted cell indicates the best result in each row. t/o = timed out after 900s.)

ity operator, which we call P_{alloyi}[4], and we evaluate each problem on increasing scopes. All models in P_{alloyi} are satisfiable with unique solutions. The models `handshake` and `magicSquare` have exact scopes and the models `square` and `account` have non-exact scopes. Further information about each model is mentioned below:

- The `handshake` model contains integers indirectly via the cardinality operator, the relational operators (`+`, `&`, `-`, `.`) and the integer operator (`=`).
- The `square` model contains a binary relation ranging over integers, the relational operators (`-`, `.`) and the integer operators (`=`, `>`, `*`). This model contains non-linear arithmetic.
- The `magicSquare` model contains a ternary relation ranging over integers, the join operator and the integer operators (`=`, `+`).
- The `account` model contains two binary relations ranging over integers, the join operator and the integer operators (`=`, `+`, `-`, `<=`, `>`).

As a whole, the models in P_{alloyi} contain integers in relation declarations of different arities and contain both arithmetic and comparison operators for integers. The models contain examples of both linear and non-linear arithmetic and a model that uses integers indirectly via the cardinality operator.

Table 6.2 demonstrates the time taken (in seconds) by each tool to solve the problems

[4]Available at https://github.com/WatForm/portus-tests.

	Command	Scope	PORTUS	Kodkod
ceilingsAndFloors	BelowToo″	120	3.42	28.72
		150	7.56	76.67
		180	11.46	132.13
lists	FalseAssertion	75	4.14	25.74
		95	8.71	66.15
		115	14.93	t/o
lights	Safe	200	42.04	21.93
		225	71.60	47.89
		250	80.19	41.75
Best out of 9			6	3
Total Time (s)			244.05	1340.98

Table 6.3: Time taken (in seconds) by PORTUS and Kodkod on P_{alloyf}
(The highlighted cell indicates the best result in each row. t/o = timed out after 900s.)

in P_{alloyi}. As expected, PORTUS performs significantly better than Kodkod except on the model containing the cardinality operator. Kodkod times out for one of the models whereas, PORTUS does not time out for any of the models in P_{alloyi}.

Finally, we choose a small subset of problems containing relations with a range of multiplicity `one`, which we call P_{alloyf}[5], and evaluate each problem on increasing scopes. All models in P_{alloyf} are unsatisfiable and have exact scopes. Further information about each model is mentioned below:

- The `ceilingsAndFloors` model contains two binary relations that are translated to functions by PORTUS and the join operator.
- The `lists` model contains two binary relations that are translated to functions by PORTUS, a signature hierarchy and the join operator.
- The `lights` model contains a ternary relation that is translated to a function by PORTUS and the relational operators (`+`, `-`, `.`, `->`).

As a whole, the models in P_{alloyf} contain relations with a range of multiplicity `one` of different arities and signature hierarchy.

Table 6.3 demonstrates the time taken (in seconds) by each tool to solve the problems in

[5]Available at https://github.com/WatForm/portus-tests.

P_{alloyf}. Portus performs better than Kodkod on the Alloy models where the optimization to replace predicates with functions is triggered. Kodkod times out for one of the models whereas, Portus does not time out for any of the models in P_{alloyf}.

6.4 Correctness Testing

The correctness of our approach is determined using extensive testing on the collection of 190 Alloy models scraped from public Github repositories. In addition to checking that Portus and Kodkod agree on the satisfiability of the models, the correctness of Portus on satisfiable models is confirmed in two ways:

- Each interpretation produced by Portus is also an instance of Kodkod.
- Each instance produced by Kodkod is also an interpretation of Portus.

In Section 4.6, we discussed the translation of a Fortress interpretation to an equivalent Kodkod instance. The Alloy Analyzer is used to translate an Alloy model to a Kodkod problem with the lower and upper bound on each relation set to exactly the tuples in the translated Kodkod instance. In order to prove that the instance is valid, Kodkod is used to solve the problem and confirm if it is satisfiable using the MiniSat SAT solver. Fortress is used to retrieve the next interpretation and the whole process is repeated again.

Similary, we map each element in Kodkod to a domain element in Fortress, reverse-engineer the predicates corresponding to each relation and add formulas for each predicate constraining its value. In order to prove that the interpretation is valid, Fortress is used to solve the problem and confirm if it is satisfiable using the Z3 SMT solver. Kodkod is used to retrieve the next instance and the whole process is repeated again.

Since proving the correctness of all interpretations of a satisfiable model is infeasible, we set a limit of five Portus interpretations and five Kodkod instances for each command of an Alloy model. Excluding the models that throw an error or timed out for either tool, Portus currently passes the correctness check on 85 satisfiable models and 57 unsatisfiable models.

6.5 Summary

This chapter evaluates the performance and correctness of Portus on a large corpus of Alloy models. The performance results demonstrate that Kodkod performs better than

Portus on models overall. After an analysis of the Alloy models, we observe that Kodkod performs better on models using relational constraints but Portus performs better on models containing integers or relation declarations with a range of multiplicity `one` (functions). The correctness of unsatisfiable models is determined by comparing the result from Portus with Kodkod. For satisfiable models, Portus cross-checks five interpretations of Portus and Kodkod.

Chapter 7

Related Work

In this chapter, we discuss previous efforts to translate Alloy to another language and relevant work that uses SMT-LIB2 to solve MSFMF problems. We start by providing details about the difference in approach compared to a previous effort to translate Alloy to Fortress called Astra. To the best of our knowledge, Portus is the only library that performs a bounded analysis with SMT-LIB2, translates all common Alloy constructs with their potential optimizations and is integrated fully with the Alloy Analyzer, along with the option to retrieve and visualize instances (or counterexamples) of satisfiable models and get the next instance.

7.1 Astra

Compared to Portus, the previous effort to translate Alloy to Fortress, Astra, uses Kodkod as an intermediary language and performs a bottom-up traversal of the Kodkod formula. In addition, Astra had many missing functionalites as mentioned in Section 2.5.

The Alloy Analyzer translates sets to unary relations and other functions and relations to n-ary relations in Kodkod using the atomization technique. Astra translates each unary relation to a sort and relations with arity greater than two to predicates or functions in Fortress. Since sets are represented by sorts and there are no membership predicates introduced, Astra can only do analysis for exact scopes. Astra introduces auxiliary functions and predicates as it traverses the formula in a bottom-up manner.

$$
\begin{aligned}
[A \cup B] &::= R_{new} : \text{TYPE}(A) \rightarrow Bool \\
&\quad \text{add } \forall x : \text{TYPE}(A) \bullet R_{new}(x) \Leftrightarrow [A](x) \vee [B](x) \\
[A \cap B] &::= R_{new} : \text{TYPE}(A) \rightarrow Bool \\
&\quad \text{add } \forall x : \text{TYPE}(A) \bullet R_{new}(x) \Leftrightarrow [A](x) \wedge [B](x) \\
[A - B] &::= R_{new} : \text{TYPE}(A) \rightarrow Bool \\
&\quad \text{add } \forall x : \text{TYPE}(A) \bullet R_{new}(x) \Leftrightarrow [A](x) \wedge \neg [B](x) \\
[A \,.\, B] &::= R_{new} : \text{TYPE}(dom(A)) \times \text{TYPE}(ran(B)) \rightarrow Bool \\
&\quad \text{add } \forall x : \text{TYPE}(dom(A)),\, y : \text{TYPE}(ran(B)) \bullet \\
&\qquad R_{new}(x, y) \Leftrightarrow \exists z : \text{TYPE}(ran(A)) \bullet [A](x, z) \wedge [B](z, y) \\
[\forall x : t \bullet A] &::= \forall x : t \bullet A \\
[v : t] &::= v : t
\end{aligned}
$$

Figure 7.1: Translation of Kodkod formulas in Astra (taken from [3])

This section discusses the motivation behind two main changes in the infrastructure of PORTUS:

1. traverse formulas in a top-down manner instead of bottom up, and
2. translate directly from Alloy to Fortress.

7.1.1 Formula Traversal

Figure 7.1 shows the translation for some set and relational operators, the universal quantifier and variables in Astra. The translation of these operators introduces auxiliary relations with additional formulas. During the bottom-up translation of quantified formulas, any relational operators generate these additional formulas, which might have an occurrence of the quantified variable in them. For instance, given a relation R of type $T_1 \rightarrow T_2$ and a quantified variable x of type T_2 in Kodkod, the set expression $R.x$ is translated as:

$$
\begin{aligned}
[R.x] \quad ::= \quad & R_{new} : T_1 \rightarrow Bool \\
& \text{add } \forall y : T_1 \bullet R_{new}(x) \Leftrightarrow \exists z : B \bullet [A](y, z) \wedge z = x
\end{aligned}
$$

Since the auxiliary formula contains the quantified variable x, it must be combined with the translation of the quantified formula in place. Putting this in general terms, for the universal quantifier, the translation of the formula results in an MSFOL formula of the form:

$$\forall x : t \bullet [A] \wedge f(x)$$

where $f(x)$ represents the conjunction of all auxiliary formulas containing an occurrence of x in them. The issue with this approach is that if Astra encounters a formula of the form:

$$\neg \ (\ \forall \ x : t \bullet A \ )$$

where A contains set operators, it would translate it to:

$$\neg \ (\ \forall \ x : t \bullet [A] \wedge f(x) \ )$$

which is equivalent to:

$$\exists \ x : t \bullet \neg \ [A] \vee \neg \ f(x)$$

resulting in erroneous results. The negation cannot be detected easily since the quantified formula can be nested and the negation may occur at any point in the bottom-up traversal.

In order to avoid this error, a possible solution is to eliminate bound variables in the resulting formulas by adding extra parameters to the auxiliary function. For the example mentioned above, $R.x$ can be translated as:

$$\begin{array}{lll} [R.x] & ::= & R_{new}(x) \text{ where } R_{new} : T_2 \times T_1 \rightarrow Bool \\ & & \text{add } \forall x_1 : B, x_2 : A \bullet R_{new}(x_1, x_2) \Leftrightarrow \exists z : B \bullet [A](x_2, z) \wedge z = x_1 \end{array}$$

For each Kodkod expression with n bound variables, the arity of the auxiliary relation and the depth of the nested universal quantification both increase by n during bound variable elimination. As we discussed in Section 2.3, the complexity of generating range formulas is exponential with respect to the arity of functions and the complexity of grounding is exponential with respect to the depth of quantification leading to an exponential increase in the solving time. In order to avoid this bottleneck in performance, the number of auxiliary functions have to be reduced which is why a top-down approach is more suitable than bottom-up.

In addition to that, the interface changes in the new Fortress make the bottom-up approach infeasible. For instance, unlike the previous version of Fortress, the new Fortress does not keep track of the sort of each term. Since Kodkod is untyped, Astra relies on Fortress to calculate the sort of a term.

7.1.2 Alloy vs. Kodkod

In our approach, we translate directly from an Alloy model to an MSFMF problem in Fortress whereas, Astra converts from Kodkod to Fortress. Since Kodkod is untyped and Fortress is sorted, Astra has to reverse-engineer the sets, set heirarchy and relations of the original model. This retrieval of information requires two passes over each Kodkod formula. We save time by acquiring this information directly from Alloy.

7.2 Other Alloy translations

El Ghazi et al. [23] reported on a case study where the assertions in a handful of Alloy models are proved using the Yices SMT solver [21] over unbounded scopes. Partial finitization is done only for the Alloy constructs using the closure operators. The translation of top-level signatures to sorts is similar to what has been done by Portus but no membership predicates are needed since the analysis is performed over unbounded domains. Since Yices allows subsorts, subsignatures are expressed using a combination of predicates and subsorts. The translation of other Alloy operators is slightly different from what is done by Portus. For instance, multiplicity keywords are expressed using auxiliary function symbols and lambda expressions as opposed to the quantified formula approach adopted by Portus. The iterative squaring approach introduced by Burch et al. [14] is used to translate the closure operators. The case study only describes the translation of limited Alloy constructs and is not complete. Although the results of the experiments seem promising, the analysis can be unsound due to the arbitrary use of quantifiers, i.e. it may produce false counterexamples.

In [24], El Ghazi et al. presented an approach to translate Alloy models to SMT-LIB2 and prove their properties using the Z3 SMT solver [20] over unbounded scopes. This approach has the most similarities with the approach discussed in Chapter 4 for Portus. Most of the Alloy operators are covered except a few including the quantified expressions, the relational operators `++`, `<:` and `:>` and the Alloy constants `univ` and `iden`. Their translation approach differs significantly in the handling of multiplicities, closure and cardinality. Multiplicity keywords in relation declarations are handled using auxiliary functions and formulas. Closure and cardinality are expressed using quantification over unbounded integers. The experimental results seem promising. However, since the Alloy logic is undecidable and there is an extensive use of quantifiers in the translation, the analysis can be unsound due to the arbitrary use of quantifiers, i.e. it may produce false counterexamples.

AlloyPE is a dual-engine framework capable of providing both counterexamples and proofs for the analysis of Alloy models [27]. AlloyPE provides three strategies: SMT-based bounded verification, SMT-based unbounded verification and ITP-based full verification using the KeY theorem prover [12]. For the SMT-based bounded and unbounded verification, the translation of an Alloy model is the same as what is shown in their previous work [24]. As discussed above, their translation, although missing a few operators, is similar to Portus except in their handling of multiplicities, closure and cardinality. For bounded verification, AlloyPE converts to quantified bit-vector formula (QBVF) logic where each sort is represented by a fixed-size bit-vector and is decidable. As shown in later work [41], AlloyPE cannot handle many specifications and fails on a considerable number of Alloy models due to unsupported Alloy constructs or internal errors during solving or translation.

A few other attempts to represent Alloy specifications in other languages include Alloy2B [34] and Alloy2JML [28] which translate Alloy specifications to the B language [4] and to the Java Modelling Language [37] respectively.

Alloy has also been paired with theorem provers to prove the properties of models. Ulbrich et al. introduced Kelloy [63], the back-engine of AlloyPE [27], which uses the KeY theorem prover [12] to provide proofs for Alloy models. Previous such tools that proved Alloy properties over unbounded scopes include Dynamite [26] which uses the PVS theorem prover [54] and Prioni [6] which uses the Athena theorem prover [1]. However, these tools are not fully automatic and require user interaction.

Alternative approaches to handle the closure and cardinality operators not explored in this thesis are mentioned here. El Ghazi et al. introduces an approach that uses invariant injections to express closure over unbounded scopes [25]. Another approach to translate closure operators on CTL-live models is introduced by Vakili et al. in [64]. Cardinality can be expressed using first-order formulas for comprehensions introduced by Leino et al. [38].

7.3 Other model finding libraries

Finite model finding libraries can be typically categorized into two classes: the MACE-style [40] and the SEM-style [71]. The MACE-style approach translates the problem to boolean logic and uses a SAT solver whereas, the SEM-style approach uses a back-tracking algorithm to explicitly search for a satisfiable interpretation. Kodkod [62, 61] is an example of a MACE-style solver that uses a SAT solver. Fortress [46, 65] is another example of a MACE-style solver. It reduces a problem to EUFBV and uses an SMT solver.

Another finite model finding library is introduced by Reynolds et al. in [51] which

converts to EUF logic to perform SEM-style model finding using an SMT solver. This library is implemented as an extension of CVC4's EUF solver [9]. However, when compared to Fortress in [65], it does not perform well.

Bansal et al. [8] introduced a new decision procedure for deciding the satisfiability of quantifier-free formulas in the theory of finite sets with membership constraints and cardinality constraints. This library is implemented in the SMT solver CVC4 [9] but is not expressive enough to be comparable to other model finding libraries discussed here.

Meng et al. [41] extends the theory of finite sets with cardinality introduced by Bansal et al. [8] to relations and relational operators including transpose, join and transitive closure. This library is also implemented in the SMT solver CVC4 [9] but cardinality constraints are left as future work. The results seem promising as compared to the AlloyPE tool [27].

AlleAlle [57] is a MACE-style relational model finding library, similar to Kodkod [62], that combines first-order logic with Codd's relational algebra [17], containing projection, restriction and join, and uses the Z3 SMT solver [20] to find solutions. Although AlleAlle provides promising results on models containing integers and cardinality compared to Kodkod, it does not perform well on models with relational operators. AlleAlle does not apply any symmetry breaking optimizations and has not been linked to Alloy yet.

Chapter 8

Conclusion

In this thesis, we presented our library, called Portus, as an alternative back-end to the Alloy Analyzer to solve Alloy models using an SMT solver. We discussed the basic rules to translate an Alloy model to an MSFMF problem in Fortress in detail. A key element in our approach is the use of set membership predicates in addition to sorts to capture the set hierarchy and to handle multiple finite scopes all within one solver problem. Portus provides support for all Alloy constructs except the bit-shifting operators and higher-order quantifications. For integers, we matched the modular arithmetic semantics in Alloy.

We identified common expressions and cases in Alloy models and suggested optimizations to improve translation in those cases. We presented a novel method to translate the cardinality operator that utilizes the built-in sort for integers and is decidable if the integers are left unbounded.

We did an extensive analysis of the performance of Portus against Kodkod. Based on our results, we have shown that Portus performs better than Kodkod on the models containing integers and relations with a range of multiplicity `one`. We determined the correctness of our method by cross-checking the instances of Portus and Kodkod. We implemented a method to simplify MSFOL formulas in the Fortress solver which improved performance.

8.1 Future Work

Although bit-shifting operators and higher-order quantifications are rarely used in Alloy models, we can add support for them in the future to make sure any models solvable by

Kodkod can also be solved by Portus. Since the modular arithmetic option for integers can produce spurious counterexamples, we plan to add support for the no overflow semantics in Fortress. Other approaches to translate cardinality and closure operators can be explored [25, 38, 64].

In the future, we plan to add more optimizations to improve the performance of Portus including the following:

1. An approach to support partial functions in SMT to increase the cases where predicates can be replaced with functions. Based on the results in Section 6.3, this particular optimization may prove very useful. This optimization will not only allow for the use of functions with non-exact scopes but will also allow us to convert Alloy relations with a range defined using multiplicity `lone` to functions in MSFOL. In addition, the predicates for *next* and *prev*, as used in the optimization of the ordering module in Section 5.6, can be represented by functions instead. Two ways to express partial functions in SMT-LIB include:

   - the use of dummy variables to handle out-of-domain values of a function, similar to what is done by Ghazi et al. in [23], but this raises the problem of adding guards on quantified variables and handling range formulas in Fortress, and
   - the use of predicates along with functions, similar to what is done by Ghazi et al. in [24], but there is doubt as to how efficient it would be.

2. Add additional symmetry breaking optimizations based on the context of the model. Examples of such optimizations include:

   - the use of domain elements to simplify constraints on the scope of subsignatures, and
   - assigning a domain using constants instead of a scope to a sort, representing a signature which has all subsignatures of multiplicity `one`.

3. Identify and remove unnecessary signatures during translation. For instance, the signatures with multiplicity `one` that are never used to bind a quantified variable can be excluded from the MSFMF problem generated by Portus. Such signatures are usually declared to introduce relations and can be identified easily by the expressions of the form `A.f` where `A` is the unnecessary signature and `f` is the relation associated with `A`. This optimization allows us to remove a sort, decrease the arity of any relations associated with that signature by one and simplify formulas. Examples of such signatures include:

- the `Ord` signature introduced by the ordering module, as mentioned in Section 5.6, and
- the `FrontDesk` signature introduced by the hotel problem to represent the relations of a single hotel in the Alloy book [31].

We plan to try other SMT solvers in Fortress. One particular solver that might give better performance is Yices [21] which allows for subtyping and lambda expressions. Since Portus provides general rules for translation from an Alloy model to an MSFMF problem, other finite model finding libraries can be connected and tried out as the backend of Portus. A few such libraries are mentioned in Section 7.3. In that direction, we also propose that Alloy should be modified to allow for easy integration of other libraries like Portus. More specifically, the dependence of the Alloy instance on the Kodkod instance can be removed by introducing a general data structure in Alloy to handle interpretations in terms of arbitrary domain elements.

References

[1] Athena. http://people.csail.mit.edu/kostas/dpls/athena/. Accessed: 2020-11-10.

[2] Web Sudoku. https://www.websudoku.com. Accessed: 2020-11-10.

[3] Ali Abbassi. Astra: Evaluating Translations from Alloy to SMT-LIB. Master's thesis, University of Waterloo, 2018.

[4] Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[5] Wilhelm Ackermann. *Solvable cases of the decision problem*. North-Holland, Amsterdam, 1956.

[6] Konstantine Arkoudas, Sarfraz Khurshid, Darko Marinov, and Martin Rinard. Integrating Model Checking and Theorem Proving for Relational Reasoning. In *International Conference on Relational Methods in Computer Science*, pages 21–33. Springer, 2003.

[7] Gilles Audemard and Laurent Simon. Glucose: a solver that predicts learnt clauses quality. *SAT Competition*, pages 7–8, 2009.

[8] Kshitij Bansal, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. A New Decision Procedure for Finite Sets and Cardinality Constraints in SMT. In *International Joint Conference on Automated Reasoning*, pages 82–98. Springer, 2016.

[9] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[10] Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[11] Jon Barwise. An introduction to first-order logic. In *Studies in Logic and the Foundations of Mathematics*, volume 90, pages 5–46. Elsevier, 1977.

[12] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of Object-Oriented Software. The KeY Approach: Foreword by K. Rustan M. Leino*, volume 4334. Springer, 2007.

[13] Armin Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. *FMV Report Series Technical Report*, 10(1), 2010.

[14] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.

[15] Koen Claessen. Expressing transitive closure for finite domains in pure first-order logic. *Unpublished draft, Chalmers University of Technology*, 2008.

[16] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation-Principles, Algorithms, Applications*, pages 11–27. Citeseer, 2003.

[17] Edgar F Codd. A Relational Model of Data for Large Shared Data Banks. In *Software pioneers*, pages 263–294. Springer, 2002.

[18] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-Breaking Predicates for Search Problems. *KR*, 96:148–159, 1996.

[19] Leonardo de Moura and Nikolaj Bjørner. Model-based Theory Combination. *Electronic Notes in Theoretical Computer Science*, 198(2):37–49, 2008.

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

[21] Bruno Dutertre and Leonardo De Moura. The YICES SMT Solver, 2006.

[22] Jonathan Edwards, Daniel Jackson, Emina Torlak, and Vincent Yeung. Faster constraint solving with subtypes. *ACM SIGSOFT Software Engineering Notes*, 29(4):232–242, 2004.

[23] Aboubakr Achraf El Ghazi and Mana Taghdiri. Analyzing Alloy Constraints using an SMT Solver: A Case Study. In *5th International Workshop on Automated Formal Methods (AFM)*, 2010.

[24] Aboubakr Achraf El Ghazi and Mana Taghdiri. Relational Reasoning via SMT Solving. In *International Symposium on Formal Methods*, pages 133–148. Springer, 2011.

[25] Aboubakr Achraf El Ghazi, Mana Taghdiri, and Mihai Herda. First-Order Transitive Closure Axiomatization via Iterative Invariant Injections. In *NASA Formal Methods Symposium*, pages 143–157. Springer, 2015.

[26] Marcelo F Frias, Carlos G Lopez Pombo, and Mariano M Moscato. Alloy Analyzer+PVS in the Analysis and Verification of Alloy Specifications. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 587–601. Springer, 2007.

[27] Aboubakr Achraf El Ghazi, Ulrich Geilmann, Mattias Ulbrich, and Mana Taghdiri. A Dual-Engine for Early Analysis of Critical Systems. *arXiv preprint arXiv:1408.0707*, 2014.

[28] Daniel Grunwald. Translating Alloy Specifications to JML. Master's thesis, Karlsruhe Institute of Technology, 2013.

[29] Daniel Jackson. An intermediate design language and its analysis. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 121–130, 1998.

[30] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[31] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT press, 2012.

[32] Eunsuk Kang and Daniel Jackson. Formal Modeling and Analysis of a Flash Filesystem in Alloy. In *International Conference on Abstract State Machines, B and Z*, pages 294–308. Springer, 2008.

[33] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. On the Complexity of Fixed-Size Bit-Vector Logics with Binary Encoded Bit-Width. In *SMT @ IJCAR*, pages 44–56, 2012.

[34] 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.

[35] Daniel Kroening and Ofer Strichman. *Decision Procedures*. Springer, 2016.

[36] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–64, 2010.

[37] Gary T Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M Zimmerman, et al. JML Reference Manual, 2008.

[38] K Rustan M Leino and Rosemary Monahan. Reasoning about Comprehensions with First-Order SMT Solvers. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 615–622, 2009.

[39] Tianhai Liu, Michael Nagel, and Mana Taghdiri. Bounded Program Verification using an SMT solver: A Case Study. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 101–110. IEEE, 2012.

[40] William McCune. A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, Argonne National Laboratory, 1994.

[41] Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. Relational Constraint Solving in SMT. In *International Conference on Automated Deduction*, pages 148–165. Springer, 2017.

[42] Aleksandar Milicevic and Daniel Jackson. Preventing arithmetic overflows in alloy. *Science of Computer Programming*, 94:203–216, 2014.

[43] Greg Nelson and Derek C Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[44] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58(4):66–73, 2015.

[45] Tina Ann Nolte. *Exploring Filesystem Synchronization with Lightweight Modeling and Analysis*. PhD thesis, Massachusetts Institute of Technology, 2002.

[46] Joseph Poremba. Static Symmetry Breaking in Many-Sorted Finite Model Finding. Undergrad thesis, 2020.

[47] Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt in Comptes Rendus du I congres de Mathématiciens des Pays Slaves. *Slaves, Warsaw*, pages 92–101, 1929.

[48] Silvio Ranise and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). http://www.smt-lib.org. 2006.

[49] Giles Reger, Martin Suda, and Andrei Voronkov. Finding Finite Models in Multi-Sorted First-Order Logic. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 323–341. Springer, 2016.

[50] Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstić. Finite Model Finding in SMT. In *International Conference on Computer Aided Verification*, pages 640–655. Springer, 2013.

[51] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters, and Clark Barrett. Quantifier Instantiation Techniques for Finite Model Finding in SMT. In *International Conference on Automated Deduction*, pages 377–391. Springer, 2013.

[52] Joanne K Rowling. *Harry Potter and the Order of the Phoenix*, volume 5. Bloomsbury Publishing, 2013.

[53] Alan B Shaffer. A Security Domain Model for Static Analysis and Verification of Software Programs. 2008.

[54] Natarajan Shankar, Sam Owre, John M Rushby, and Dave WJ Stringer-Calvert. PVS Prover Guide. *Computer Science Laboratory, SRI International, Menlo Park, CA*, 1:11–12, 2001.

[55] Ilya Shlyakhter. *Declarative Symbolic Pure-Logic Model Checking*. PhD thesis, Massachusetts Institute of Technology, 2005.

[56] Niklas Sorensson and Niklas Een. MiniSat v1.13 – A SAT Solver with Conflict-Clause Minimization. *SAT*, 2005(53):1–2, 2005.

[57] Jouke Stoel, Tijs van der Storm, and Jurgen J Vinju. AlleAlle: Bounded Relational Model Finding with Unbounded Data. In *Proceedings of the 2019 ACM SIGPLAN*

International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, pages 46–61, 2019.

[58] Geoffrey Sutcliffe. The TPTP problem library and associated infrastructure: from CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, pages 1–20, 2017.

[59] Mana Taghdiri and Daniel Jackson. A Lightweight Formal Analysis of a Multicast Key Management Scheme. In *International Conference on Formal Techniques for Networked and Distributed Systems*, pages 240–256. Springer, 2003.

[60] Tanel Tammet. Gandalf. *Journal of Automated Reasoning*, 18(2):199–204, 1997.

[61] Emina Torlak. *A constraint solver for software engineering: finding models and cores of large relational specifications*. PhD thesis, Massachusetts Institute of Technology, 2009.

[62] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.

[63] Mattias Ulbrich, Ulrich Geilmann, Aboubakr Achraf El Ghazi, and Mana Taghdiri. A Proof Assistant for Alloy Specifications. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 422–436. Springer, 2012.

[64] Amirhossein Vakili and Nancy A Day. Reducing CTL-Live Model Checking to First-Order Logic Validity Checking. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 215–218. IEEE, 2014.

[65] Amirhossein Vakili and Nancy A Day. Finite Model Finding Using the Logic of Equality with Uninterpreted Functions. In *International Symposium on Formal Methods*, pages 677–693. Springer, 2016.

[66] Jan Van Eijck. Defining (reflexive) transitive closure on finite models, 2008.

[67] Christoph M Wintersteiger. *Termination Analysis for Bit-Vector Programs*. PhD thesis, ETH Zurich, 2011.

[68] Christoph M Wintersteiger, Youssef Hamadi, and Leonardo De Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23, 2013.

[69] John Zao, Hoetech Wee, Jonathan Chu, and Daniel Jackson. RBAC Schema Verification using Lightweight Formal Model and Constraint Analysis. *Submitted to SACMAT*, 2003.

[70] Pamela Zave. Using Lightweight Modeling to Understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2):49–57, 2012.

[71] Hantao Zhang and Jian Zhang. MACE4 and SEM: A Comparison of Finite Model Generators. In *Automated Reasoning and Mathematics*, pages 101–130. Springer, 2013.

APPENDICES

Appendix A

Tool versions

This appendix lists the version of each tool used:

- Alloy Analyzer: v5.1.0
- MiniSat: v2.2.0
- Z3: v4.8.8
- CVC4: v1.8

Appendix B

BNF operators

This appendix lists the operators used by the Backus-Naur form (BNF), the notation used to specify the abstract syntax of languages:

- x^* for zero or more repititions of x
- x^+ for one or more repititions of x
- $x \mid y$ for a choice x or y
- $[x]$ for an optional x

Appendix C

Optimizations

Table C.1 demonstrates the profiling characteristics for each closure approach on the models not included in Section 5.7 in the interest of space.

	addressBook2e				
	Scope = 7				
	basic	Burch et al.	Claessen	Eijck	Liu et al.
Functions	0	0	1	0	1
Predicates	8	10	10	9	8
Max Arity	3	3	4	4	3
Depth Quant	9	7	7	7	7
Skolem Functions	2	4	2	3	3
Initial Term Count	×1	×0.71	×0.89	×0.87	×0.74
Final Term Count	×1	×0.002	×0.005	×0.005	×0.001
Transform Time (s)	20.2	0.14	0.27	0.29	0.10

	grandpa				
	Scope = 20				
	basic	Burch et al.	Claessen	Eijck	Liu et al.
Functions	0	0	1	0	1
Predicates	7	11	9	8	8
Max Arity	2	2	3	3	3
Depth Quant	21	3	4	4	7
Skolem Constants	-	1	1	1	7
Skolem Functions	-	6	2	3	3
Initial Term Count	×1	×0.14	×0.15	×0.15	×0.13
Final Term Count	-	×1	×3.81	×3.93	×0.15
Transform Time (s)	-	0.25	1.07	1.19	0.10

Table C.1: Profiling characteristics for comparison of closure translations in Portus