Improvements to Many-Sorted Finite Model Finding using SMT Solvers

by

Owen Zila

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, 2023

© Owen Zila 2023

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

Formal modeling is a powerful tool in requirements engineering. By modeling a system before implementation, one can discover bugs before they appear in testing or production. Model finding (or instance finding) for a model written in first-order logic is the problem of finding a mapping of variables to values that satisfies a model's specifications. Unfortunately, this problem is undecidable in general. Finite model finding, the problem of finding a mapping of variables to finite sets of values, is decidable. Thus, finite model finding enables the automated verification of models at certain scopes (the number of elements in the domain of discourse of the problem). While finitizing a problem makes it solvable, as the scope of the problem increases, the problem can quickly become prohibitivly expensive to solve. Therefore, it is important to choose an efficient encoding of the problem for satisfiability (SAT) or satisfiability modulo theories (SMT) solvers when finitizing a problem.

We propose improvements to encodings of many-sorted finite model finding problems for SMT solvers. We propose new encodings for finite integers and transitive closure. The key contributions of this thesis are that we:

1. Formulate Milicevic and Jackson's [25] method for preventing overflows in integer problems using bitvectors as a transformation from/to a many-sorted first-order logic formula and extend it to support additional abstractions present in MSFOL

2. Propose an integer finitization method, called overflow-preventing finite integers (OPFI), that produces results closer to unbounded integers than Milicevic and Jackson's method, improving the correctness of the finitization with respect to the same problem over unbounded integers

3. Demonstrate that OPFI solves problems faster than our encoding of Milicevic and Jackson's method in an SMT solver, and does not solve problems significantly slower than unchecked (pure) bitvectors

4. Propose and prove the validity of negative transitive closure, an encoding of the transitive closure operator over a finite scope in first order logic for a special case of the use of transitive closure where pairs are only checked to not be in the transitive closure of a relation

5. Generalize existing encodings of transitive closure to relations of arity greater than two

6. Demonstrate our new encoding of transitive closure performs faster than or as fast as existing encodings on models generated from Alloy models

Acknowledgements

I would like to thank my supervisor, Prof. Nancy A. Day, whose support and guidance was instrumental throughout my Master's program. Prof. Day's care, support, patience, and encouragement was instrumental in the writing of this thesis. While completing this thesis, I was halfway across the world with a terrible internet connection, but Nancy supported me every step of the way. I will never be able to thank her enough for all the support and patience she has given me.

I would also like to thank my second readers, Joanne Atlee and Derek Rayside. Their comments and insight were invaluable.

I would also like to express my gratitude to my colleagues. I was helped immensely by Aditya Shankar Narayanan and Ryan Dancy's work on Dash+ and Portus respectively. Both of them were incredibly helpful and responsive to any and all questions and issues.

Finally, I would like to extend my thanks to my friends and family for their endless kindness and support throughout my life and especially the past few months.

Dedication

To my mom, dad, and brother, whose love and support I feel every day.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Formal modeling, which is the description of system behavior in a mathematically precise way, is a powerful tool in requirements engineering [5]. Projects employing robust software engineering, of which modelling is an important factor, are more likely to be delivered successfully, on-time, and on-budget [14]. By modeling a system before implementation, one can discover bugs before they appear in testing or production. For example, model checking discovered five errors (including identifying major design flaws) in an execution module of the Deep Space 1 spacecraft and revealed design flaws in the software controlling storm surge barriers in Rotterdam [5]. Unfortunately, the verification of an arbitrary model is undecidable [10]. When a model is finite, however, it can be verified by an exhaustive search, making the problem decidable. The process of finding a finite mapping of variables to values that satisfies the finite model's specification is known as finite model finding (FMF). So, altering a problem to be finite allows us to search a bounded model for inconsistencies and design flaws.

When finitizing a problem, one runs into several challenges. When a problem is finitized, its meaning may change. Statements of the form "For all ..." become akin to "For all values between 0 and 9 ..." instead. Notably, while finitizing a problem makes it solvable, as the scope of the problem (the number of elements the model considers) increases, the problem can quickly become prohibitively expensive to solve [18]. Daniel Jackson's small scope hypothesis states that most issues with a model can be found at smaller, finite scopes [18]. Thus, these semantic changes to quantifiers are likely acceptable for the verification of many applications. However, it is important when finitizing a problem to choose an efficient encoding for solvers.

Fortress is a method/library that leverages satisfiability modulo theories (SMT) solvers

for examining the validity of finite models. Fortress represents problems in many-sorted first-order logic (MSFOL), where the domain of discourse is separated into disjoint sets of values known as sorts. FMF for many-sorted logic is known as many-sorted finite model finding (MSFMF). Vakili and Day originally implemented Fortress to encode finite model finding in the logic of equality and uninterpreted functions (EUF) [30].

The goal of this thesis is to investigate optimizations and generalizations for MSFMF to improve performance and the correctness of various axiomatizations. We implement our work in Fortress. Finitizing integers is complicated by the many interpreted functions on integers (+, -, *, etc.). Operations on finitely bounded integers overflow when the expected result is not within the expected range. When a problem is finitized, the size of the problem may be prohibitive to efficient solving [10], so integer ranges cannot simply be increased to avoid overflows occurring. Milicevic and Jackson proposed a solution to this problem and implement it in the Kodkod solver for Alloy using a SAT solver [25]. They finitely represented integers as bitvectors and altered the semantics of quantifiers and predicates to "ignore" overflows. In the first half of this thesis, we propose a new method for finitizing integers and handling overflows that produces results theoretically closer to unbounded integers by taking advantage of an SMT solver's interpretation of unbounded integers. Our method is also experimentally faster at solving problems in MSFOL.

The second half of the thesis explores various axiomatizations of transitive closure. We examine prior work on encoding transitive closure in MFSOL. Then, we propose a new axiomatization for transitive closure under specific conditions that requires no auxiliary functions. We then propose a generalization for these encodings for closure over relations of arity greater than two. We evaluate the performance of these axiomatizations at various scopes.

1.1 Contributions

In this thesis, we:

1. Formulate Milicevic and Jackson's [25] method for preventing overflows in integer problems using bitvectors to represent integers as a transformation from/to an MSFOL formula and we extend Milicevic and Jackson's method to support additional abstractions present in MSFOL

2. Propose an integer finitization method that overflows less often than bitvectors, improving the correctness of the finitization with regard to the same problem over unbounded integers

3. Compare the performance of various integer finitization methods

4. Propose and prove the validity of negative transitive closure, an encoding of the transitive closure operator over a finite scope in first order logic for a special case of the use of transitive closure where pairs are only checked to not be in the transitive closure of a relation

5. Generalize existing encodings of transitive closure to relations of arity greater than two

6. Compare the performance our encodings of transitive closure

1.2 Outline of Thesis

Chapter 2 provides background on many-sorted first-order logic (MSFOL), satisfiability modulo theories (SMT), many-sorted finite model finding (MSFMF), existing modelling tools, and Fortress. Chapter 3 provides a description of Milicevic and Jackson's overflow prevention for bitvectors as a transformation of MSFOL formulas, our new method for integer finitization, and a performance comparison of various integer reasoning methods on the SMT-LIB UFNIA benchmark. Chapter 4 describes existing methods for axiomatizing transitive closure in MSFMF problems, proposes our new transitive closure axiomatization, generalizes both to higher-arity relations, and compares their performance. We describe related work in each of Chapter 3 and Chapter 4 as needed.

Chapter 2

Background

In this chapter, we introduce the terminology used in many-sorted first-order logic (MSFOL) and the notation we will use throughout this thesis. Then, we provide a background on satisfiability modulo theories, many-sorted finite model finding (MSFMF), and the Fortress library.

2.1 Many-Sorted First-Order Logic (MSFOL)

Many-sorted first-order logic (MSFOL) is first-order logic (FOL) but the domain of discourse is partitioned into disjoint sets called sorts. Terms in MSFOL are composed of functional symbols (f_i), variables (v_i), and quantifiers ($\forall$ and $\exists$, which are universal and existential respectively). The domain of discourse can change from problem to problem, with each problem using different sorts and functional symbols. A signature, Σ, is a tuple $(\Theta, \mathcal{F})$ where:

1. Θ is a finite set of symbols called sorts.
2. $\mathcal{F}$ is a set of functional symbols of the form $f : A_1 \times \cdots \times A_n \to B$ where $A_1 \ldots A_n, B$ are all sorts in Θ.

A functional symbol's arity is its number of arguments. Given a functional symbol $f : A_1 \times \ldots \times A_n \to B$, one says n is the arity of f. Constants are functional symbols with arity 0. A constant c of sort A is written $c : A$.

Additionally, the Boolean sort $Bool$, the set of logical connectives, and the Boolean constants are included in all MSFOL signatures. The sort $Bool$ represents Boolean values with constants true ($\top$) and false ($\bot$). The logical connectives not ($\neg$), and ($\wedge$), or ($\vee$), implies ($\Rightarrow$), and iff ($\Leftrightarrow$), are all $Bool \times Bool \rightarrow Bool$ except $\neg : Bool \rightarrow Bool$. A predicate p is any functional symbol of the form $p : A_1 \times \cdots \times A_n \rightarrow Bool$. All of the logical connectives are predicates. We also include the equality operator ($=_A$: $A \times A \rightarrow Bool$) for every sort A. Typically, the subscript is omitted and the equality operator is written as simply $=$.

A term ϕ over a signature $\Sigma = (\Theta, \mathcal{F})$ in MSFOL can be constructed as follows:

$$\begin{aligned}
\phi &= v \quad \text{where } v \text{ is a variable} \\
&= c \quad \text{where } c \text{ is a constant in } \mathcal{F} \\
&= f(\phi_1, \ldots \phi_n) \quad \text{where } f \text{ is a function of arity } n \text{ in } \mathcal{F} \\
&= \exists v : A \bullet \phi \quad \text{where } v \text{ is a variable and } A \text{ is a sort in } \Theta \\
&= \forall v : A \bullet \phi \quad \text{where } v \text{ is a variable and } A \text{ is a sort in } \Theta
\end{aligned}$$

A term in MSFOL must be well-sorted, meaning that functional symbols can be applied only to arguments of the correct sort and the arguments to the equality operator must be of the same sort. A formula in MSFOL is a term of sort $Bool$.

When the sort of a variable in $\exists v : A \bullet \phi$ or $\forall v : A \bullet \phi$ is known, the sort of v can be omitted, allowing the term to be written as $\exists v \bullet \phi$ or $\forall v \bullet \phi$. Often, the application of a binary functional symbol is written infix (such as $a \vee b$ or $a = b$ instead of $\vee(a, b)$ or $= (a, b)$), while negation is often written without parenthesis (such as $\neg a$ instead of $\neg(a)$). The term $\phi\,[x|y]$ refers to the term ϕ where every occurrence of x is replaced with y.

The free variables in a term refer to the set of variables that are not quantified. The free variables of a term consisting of v is $\{v\}$, the free variables of $f(\phi_1, \ldots, \phi_n)$ is the union of the free variables in $\phi_1, \ldots, \phi_n$, and the free variables in $\exists x \bullet \phi$ or $\forall x \bullet \phi$ is the set of free variables in ϕ excluding x. A term is closed if it contains no free variables.

A scope assignment for a signature $\mathcal{S} : \Theta \rightarrow Int$ is a mapping from each sort $\theta \in \Theta$ to its scope $k \geq 1$. The scope of a sort is the number of distinct values in the domain of the sort $\mathcal{D}_\theta$. Semantically, quantifiers quantify over the domain of a sort. Sorts cannot be empty.

The semantics of an MSFOL term are determined by interpretations. An interpretation $\mathcal{I}$ of a signature $\Sigma = (\Theta, \mathcal{F})$ with scope assignment $\mathcal{S}$ is a mapping that assigns values to an MSFOL signature, where:

1. every constant $(c : A) \in \mathcal{F}$ is assigned an element, $c^{\mathcal{I}}$ in $\mathcal{D}_A$
2. every functional symbol $(f : A_1 \times \cdots \times A_n \to B) \in \mathcal{F}$ of arity $n > 0$ is assigned a total function, $f^{\mathcal{I}}$, over the domains of its sorts, $f^{\mathcal{I}} : \mathcal{D}_{A_1} \times \cdots \times \mathcal{D}_{A_n} \to \mathcal{D}_B$.

Let $\mathcal{I}\,[x|v]$ be a mapping that contains all of the mappings in $\mathcal{I}$ as well as a mapping from x to value v. In the case where $x^{\mathcal{I}}$ is already mapped to a value, the previous value of x is discarded.

Using interpretations, we can define the semantics of every closed MSFOL term. We overload the equality operator $=$ for use on values as well as terms. So, $a^{\mathcal{I}} = v$ says the value that the term a is mapped to by interpretation $\mathcal{I}$ is equal to value v. An interpretation maps term ϕ to value $\phi^{\mathcal{I}}$, where:

1. Any constant c is mapped to $c^{\mathcal{I}}$
2. Function invocations $f(x_1, \ldots, x_n)^{\mathcal{I}} = f^{\mathcal{I}}({x_1}^{\mathcal{I}}, \ldots, {x_n}^{\mathcal{I}})$
3. The $\forall$ quantifier is interpreted as

$$(\forall x : A \bullet \phi)^{\mathcal{I}} = \bigwedge_{v \in \mathcal{D}_A}^{\mathcal{I}} \phi^{\mathcal{I}[x|v]}$$

4. The $\exists$ quantifier is interpreted as

$$(\exists x : A \bullet \phi)^{\mathcal{I}} = \bigvee_{v \in \mathcal{D}_A}^{\mathcal{I}} \phi^{\mathcal{I}[x|v]}$$

5. The Boolean constants have the same values in every interpretation: $\top^{\mathcal{I}} = true$ and $\bot^{\mathcal{I}} = false$
6. The logical connectives are the same in all interpretations and have their usual meaning. For example,

$$(\neg x)^{\mathcal{I}} = \begin{cases} true & x^{\mathcal{I}} = false \\ false & \text{otherwise} \end{cases}$$

$$(x \wedge y)^{\mathcal{I}} = \begin{cases} true & x^{\mathcal{I}} = true \text{ and } y^{\mathcal{I}} = true \\ false & \text{otherwise} \end{cases}$$

7. Equality is defined as
$$\left((x = y)^{\mathcal{I}}\right) \;=\; \left(x^{\mathcal{I}} =^{\mathcal{I}} y^{\mathcal{I}}\right)$$
where x and y have the same sort.

As logical connectives and the equality operators have the same value regardless of the interpretation, we will often omit the $\cdot^{\mathcal{I}}$ in this thesis, writing $a \wedge^{\mathcal{I}} b$ as $a \wedge b$ and $a =^{\mathcal{I}} b$ as $a = b$ for example.

An interpretation $\mathcal{I}$ of signature Σ under scope assignment $\mathcal{S}$ satisfies or "models" a formula ϕ, written $\mathcal{I} \Vdash \phi$, if and only if $\phi^{\mathcal{I}} = true$. An interpretation satisfies a set of formulas Γ, written $\mathcal{I} \Vdash \Gamma$, if and only if $\forall \phi \in \Gamma \bullet \mathcal{I} \Vdash \phi$. A formula, or set of formulas, is satisfiable (SAT) if there exists some interpretation that satisfies it. If a formula, or set of formulas, is not satisfiable, it is unsatisfiable (UNSAT).

Two formulas ϕ and ψ are equisatisfiable, denoted $\phi \overset{\text{SAT}}{=} \psi$, if either both or neither ϕ and ψ are satisfiable. An interpretation that satisfies ϕ does not need to satisfy ψ or vice versa for ϕ and ψ to be equisatisfiable, nor do ϕ and ψ need the same number of satisfying interpretations.

A set of formulas Δ is valid with respect to a set of formulas Γ (written $\Gamma \models \Delta$) if and only if for every $\mathcal{I}$ that satisfies Γ, $\mathcal{I}$ satisfies Δ.

An MSFOL problem $\mathcal{P}$ is a tuple of a signature Σ, a finite set of MSFOL formulas Γ, and a domain assignment $\mathcal{S}$. Sometimes, as a part of a problem, functional symbols are defined by a body. These are referred to as interpreted functions (while the rest are uninterpreted). If a problem includes an interpreted function $f(p_1 : A_1, \ldots, p_n : A_n) \to B = \phi$,where ϕ is a term of sort B,
$$(f(a_1, \ldots, a_n))^{\mathcal{I}} = (\phi\,[a_1, \ldots, a_n | p_1, \ldots, p_n])^{\mathcal{I}}$$
in order for $\mathcal{I}$ to be a satisfying interpretation for the problem.

2.2 Satisfiability Modulo Theories (SMT)

Satisfiability Modulo Theories (SMT) is the problem of determining if a first-order formula is satisfiable with regards to one or more specific restrictions to the semantics of certain symbols in an interpretation [4]. Each of these specific restrictions is called a background theory. Theories are sets of interpreted functions. Chapter 3, for example, deals with theories of integer arithmetic, bitvectors, and uninterpreted functions. SMT is decidable

for terms within some fragments of MSFOL, while others are undecidable. For example, the theory of quantifier-free uninterpreted functions with equality (EUF) is decidable [1], while the theory of arrays with nested reads of the form $a_1[a_2[i]]$ is undecidable if i is universally quantified [6]. An SMT solver is a tool for solving SMT problems.

The SMT-LIB standard defines a language for writing MSFOL formulas, background theories, and a command language for querying SMT solvers and examining the results [2]. SMT-LIB is also an international initiative to collect a large library of benchmarks for SMT problems, as well as standardizing descriptions of background theories [3].

As MSFOL is generally undecidable, SMT solvers may not be able to determine if a problem is SAT or UNSAT. The SMT-LIB standard allows for the response `unknown` when a solver fails to determine if a problem is SAT or UNSAT.

2.3 Many-Sorted Finite Model Finding (MSFMF)

Given an MSFOL problem $\mathcal{P}$, the problem of MSFMF is to find an finite interpretation that satisfies every formula in the set. In an MSFMF problem, the scope of every sort is finite. This is the many-sorted version of version of (single-sorted) finite model finding (FMF). A practice called symmetry reduction is used to improve performance by avoiding considering interpretations that are isomorphic (symmetric) to other interpretations [9].

MSFMF is typically solved in one of two ways. MACE-style solvers such as MACE2 [23], Paradox [9], Kodkod [29], and Vampire [20] reduce the problem to Boolean satisfiability and use a SAT solver to solve the problem. MACE-style solvers must statically encode their knowledge of the problem, including symmetry breaking, before applying the solver to it. SEM-style solvers such as SEM [33] and MACE4 [24] explore possible interpretations directly and recursively backtrack to find a satisfying interpretation. This exploration of the problem allows SEM-style solvers to dynamically use symmetry breaking during solving.

Some solvers, such as Paradox [9] and Kodkod [29] are unsorted (or single-sorted), while others (such as SEM [33] and Vampire [20]) fully utilize multiple sorts. Alloy [18] is a popular modelling language based on FMF using the Kodkod solver.

2.4 Fortress

Fortress is a method/library for solving MSFMF problems using an SMT solver. It is a MACE-style solver, as it reduces the problem to logic, then uses an off-the-shelf SMT solver to solve the problem. It is distinct from other MSFMF solvers in that the MSFMF problem is captured in the equality of uninterpreted functions (EUF) rather than propositional logic. EUF is decidable [4].

Fortress transforms a MSFMF problem to an equisatisfiable problem in MSFOL. Our presentation of Fortress follows Vakili and Day's presentation [31]. Consider the problem where $\Sigma = (\{A, B\}, \{f : A \to B\})$ is a signature, the set of formulas Γ contains only the formula

$$\forall x, y : A \bullet f(x) = f(y) \;\Rightarrow\; x = y \tag{2.1}$$

and the scope assignment for A is 3 and B is 2. Equation 2.1 means f is injective, so $|A| \leq |B|$. Therefore, this problem is UNSAT for this scope assignment. To reduce the problem to EUF, we introduce distinct constants for each scope value to represent the elements in the domain. We create $a_1, a_2, a_3 : A$ and $b_1, b_2 : B$. To ensure every interpretation assigns each of these constants a different value, we add a set of constraints to ensure the constants are distinct:

$$\{a_1 \neq a_2, a_1 \neq a_3, a_2 \neq a_3, b_1 \neq b_2\} \tag{2.2}$$

and expand the remaining universal quantifiers, replacing Equation 2.1 with

$$\bigwedge_{1 \leq x, y \leq 3} f(a_x) = f(a_y) \;\Rightarrow\; a_x = a_y \tag{2.3}$$

However, this is not an adequate reduction yet, as an instance with three domain elements for B would be a satisfying instance for Equation 2.2 and Equation 2.1. Equation 2.2 ensures that $|A| \geq 3$ and $|B| \geq 2$ for any satisfying instance, but we have not added a constraint for the maximum scope of the sorts. Adding a formula such as

$$\forall b : B \bullet b = b_1 \vee b = b_2$$

is insufficient, as expanding the universal quantifier transforms the formula to

$$(b_1 = b_1 \vee b_2 = b_1) \wedge (b_2 = b_1 \vee b_2 = b_2)$$

which is a tautology. Instead, we add constraints called range formulas that restrict the output of functions so that their results are always one of the domain constants. In this example, we add

$$\forall a : A \bullet f(a) = b_1 \vee f(a) = b_2$$

Expanding the quantifier in this equation yields

$$\bigwedge_{1 \leq x \leq 3} f(a_x) = b_1 \vee f(a_x) = b_2$$

Range formulas ensure that the interpretation of any term or subterm in the set of formulas must be one of the domain constants we enumerated so any additional domain elements cannot affect the satisfiability of Γ.

Fortress then translates the input problem to the SMT-LIB standard version 2.6, allowing it to use a variety of off-the-shelf SMT solvers that support the standard. Currently, Fortress uses Z3, an SMT solver developed by Microsoft Research [12].

Fortress is implemented as a Scala library that uses SMT solvers to check the satisfiability of MSFMF problems [28]. Fortress accepts input via its API, in Scala and Java, and a subset of the SMT-LIB standard version 2.6 (mostly excluding commands for SMT solvers). Fortress then finitizes the problem by applying a series of transformers (referred to collectively as a compiler), which are transformations from and to MSFMF problems. Generally, problems are first typechecked, then put into negation normal form (NNF) before a transformer is applied to skolemize the formulas in the problem, eliminating existential quantification. Symmetry breaking is then introduced to avoid considering equivalent interpretations. Then, universal quantifiers are expanded into a conjunction of terms, before range formulas are added to the problem, resulting in the problem being in the logic of equality and uninterpreted functions (EUF). The problem is then simplified to remove terms that will always evaluate to true or false. Domain elements are encoded in SMT-LIB as either distinct constants or as values in an algebraic datatype. In this thesis, evaluations use the distinct constants method. Poremba et al. extended Fortress to support both unbounded and finite, bitvector implementations of integers [26]. Fortress includes the logic of bitvectors (BV) or integers, when the input problem contains bitvectors or integers respectively. These logics are outside of EUF, though BV is still decidable [4].

Fortress can be extended by adding transformers and including them in compilers. For example, in this thesis, we add transformers to finitize integers and expand transitive closure.

Chapter 3

Integer Reasoning

Integers are a common construct used in modeling. Over 35% of Alloy models contain integers [**?**]. Finite reasoning about numbers within an SMT solver can be distinct from reasoning about other uninterpreted sorts because SMT solvers include theories for unbounded integers. However, these theories can decide only specific subsets of integer problems (e.g., linear integer arithmetic [4]). Outside of these specific types of integer problems, the solver is not guaranteed to be able to determine if a problem is satisfiable or not. To avoid this issue, a problem with unbounded integers can be changed into one over a finite set of integers. In FMF solvers, such as Kodkod, where the entire problem is mapped to propositional logic for a SAT solver, the backend solver does not have any understanding of integers and thus encoding integers becomes part of the process of encoding any finite sort. The question that we examine in this chapter is how to take advantage of the integer reasoning engines provided within an SMT solver for reasoning about finite number sets in problems in Fortress.

A common approach to finitely representing integers is to use fixed-width bitvectors. Bitvectors can represent a finite range (typically $\left[-2^{b-1}, 2^{b-1}-1\right]$ where b is the bitwidth), making the problem finite, and therefore decidable. We call this method unchecked bitvectors (UBV). However, because they are finite, bitvectors behave differently from unbounded integers. Specifically, operations can "overflow". For example, if the bitvectors are of width 3 (representing the finite range $[-4, 3]$), then you cannot represent $3+3$ as a bitvector because 6 is not within range. The result of every operation must be a value within range. Kroening and Strichman [21] point out that bitvector overflow can alter how seemingly obvious formulas behave. They give the example

$$\forall x, y \bullet (x - y > 0) \Leftrightarrow (x > y) \tag{3.1}$$

which does not hold over finite-width bitvectors due to overflow in the the subtraction operation. For example, over $[-4, 3]$, the assignment $\{x = -4, y = 1\}$ would cause $x > y$ to be false, but $-4 - 1 > 0$ might evaluate to $3 > 0$ (as -5 is out of range) and then the antecedent is true, making Equation 3.1 false.

Milicevic and Jackson [25] give an alternative semantics to quantification over integers, integer predicates, and integer functions that "ignore" subterms that contain overflows. For example, if a subterm contains universally quantified variables, the predicate is made to be true when an overflow occurs. Otherwise, the predicate is false. As another example, for the formula

$$\exists x \bullet x + 10 \leq x \tag{3.2}$$

when evaluating the formula over the integers $[-16, 15]$, an overflow allows $x = 6$ to be a valid solution (due to wraparound semantics where 16 would become -16). Under Milicevic and Jackson's semantics, $x + 10 \leq x$ evaluates to false because $6 + 10$ is not representable (causing an overflow) and x is existentially quantified. After considering all non-overflowing values for x, the problem unsatisfiable, as one would want. This result matches the formulas meaning in unbounded integers, because no assignment for x exists such that the formula evaluates to true.

However, when overflows occur, a formula under Milicevic and Jackson's method can evaluate to true or false when evaluating the same instance over unbounded integers would give the opposite result. For the sake of our discussion in this chapter, we call the validity/satisfiability of a formula over unbounded integers the *correct* result. The correctness we are considering is of the finitization method, not of the solver's behavior on the finitized problem. If an incorrect result is obtained, it is assumed to be the result of the finitization method, rather than an error in the solver.

Consider the formulas[1]

$$\exists x : Int \bullet x + 100 - 100 = x \tag{3.3}$$
$$\forall x : Int \bullet x + 100 - 100 \neq x \tag{3.4}$$

over the range $[-16, 15]$. For every x in that range, the expression $x + 100$ causes an overflow. Thus, this method will "ignore" every assignment to x because the $x + 100$ subterm overflows. Therefore, Equation 3.3 evaluates to false and Equation 3.4 evaluates to true; in both cases the evaluation under Milicevic and Jackson's method is incorrect with regards to solving the problem over unbounded integers. In this chapter, we present a new method, called overflow-preventing finite integers (OPFI), which overflows less often;

[1] We show a more complicated example later that cannot be fixed through simplification of the term.

intermediate terms' values are allowed to go outside the range without being considered an overflow. For example, Equation 3.3 is true and Equation 3.4 is false in our method. Our method reduces the number of MSFMF problems where these incorrect results occur.

Section 3.1 discusses the formulation of Milicevic and Jackson's method at the MSFOL level including extensions to support additional abstractions present in MSFOL. We will refer to our formulation of Milicevic and Jackson's method as No-Overflow Bitvectors (NOBV). Section 3.2 presents Overflow-Preventing Finite Integers (OPFI), a new method for finitizing integer MSFMF problems containing integers. Section 3.3 presents a set of examples concerning the correctness of the integer finitization methods considered on various problems, followed by a performance evaluation of the methods.

This chapter discusses three contributions.

1. We formulate Milicevic and Jackson's method as NOBV, a transformation of an MSFOL formula using bitvectors for integer representation to another MSFOL formula using bitvectors for a comparison with other methods and extend NOBV to support additional abstractions present in MSFOL (Section 3.1).

2. We provide a new method for finitizing a MSFMF problem containing integers, called Overflow-Preventing Finite Integers (OPFI), that utilizes an SMT solver's ability to represent arbitrary integers. OPFI theoretically gives more accurate SAT and UNSAT results than NOBV (Section 3.2).

3. We present a performance evaluation of NOBV, OPFI, and unchecked bitvectors (UBV) on the SMT-LIB UFNIA benchmark set (Section 3.3). Our analysis shows OPFI is faster than NOBV when using an SMT solver.

3.1 No-Overflow Bitvectors (NOBV)

Milicevic and Jackson's approach is applicable to integers that are already represented as bitvectors. They describe their method at two levels: a higher-level three-valued logic, where overflows create unknown values; and a lower-level evaluation semantics for a problem in propositional logic [25]. In both cases, integers are already represented by bitvectors. Fortress transforms formulas from/to many-sorted first-order logic, which it then sends to an SMT solver enabling us to use multiple sorts and MSFOL features such as uninterpreted functions within formulas. Thus, we reformulate and generalize Milicevic and Jackson's method as a transformation of a problem in MSFOL with bitvectors. We call our reformulation of Milicevic and Jackson's method "no-overflow bitvectors" (NOBV).

In both NOBV and Milicevic and Jackson's method the formula is already using bitvectors to represent integers. In Milicevic and Jackson's evaluation semantics, integer predicates are augmented with checks *at the SAT level* to determine if any of the subterms overflowed, thus changing bitvector semantics. Milicevic and Jackson implement their method in Kodkod, by transforming their input formula into a digital circuit and adding additional overflow circuits to integer operations. The most notable change we make to Milicevic and Jackson's method is that we do not change bitvector semantics. We send our results to an SMT solver that comes with its own theory for bitvectors, which we cannot modify. Therefore, where Milicevic and Jackson describe a term overflowing as it *evaluating* to an unknown value (due to overflowing), we instead include the predicate $\textsc{overflows}(f(args))$ statically within a formula. $\textsc{overflows}(f(args))$ evaluates to true if and only if the arguments to f would cause $f(args)$ to overflow. For example, for a range of $[-16, 15]$, $\textsc{overflows}(-x) \equiv x = -16$ as the negation of -16 is out of range. We implement $\textsc{overflows}(a * b)$ by zero-extending a and b and checking if their product is out of bounds. Integer division will not overflow, but is undefined for $x/0$, so $\textsc{overflows}(a/b) = (b = 0)$.

In NOBV, when an integer predicate applied to terms contains a subterm that overflows, it is replaced with a Boolean value to "ignore" the result of this predicate application so the quantifier ignores the assignment causing an overflow. In

$$\exists x \bullet x + 10 \leq x \qquad \text{(3.2 revisited)}$$

the term $x + 10 \leq x$ should be false when $x + 10$ overflows, so that the entire formula does not incorrectly become true. By including $\textsc{overflows}(x + 10)$, we obtain the formula

$$\exists x \bullet x + 10 \leq x \wedge \neg \textsc{overflows}(x + 10)$$

which correctly evaluates to false for integers over any finite range. If the overflowing term is universally quantified, the predicate is made true. Otherwise, such as in the above example, the predicate is made false.

However, when terms are negated, choosing a Boolean value based solely on how the overflowing term is quantified gives incorrect results. Consider the formula

$$\exists x \bullet \neg(x + 10 > x)$$

which is equivalent to Equation 3.2. If $x + 10$ overflows and $x + 10 > x$ is made false, the entire formula incorrectly evaluates to true.

Milicevic and Jackson recursively considered the polarity of a formula with regards to its parent formula to correctly evaluate negated formulas. The polarity of a top-level formula

is positive. Any time a subterm is negated (i.e., inside the $\neg$ operator), the polarity of the subterm is flipped. For example, given the formula $\neg\phi$, ϕ has the opposite polarity of $\neg\phi$. In $\phi \vee \psi$ and $\phi \wedge \psi$ both ϕ and ψ have the same polarity as $\phi \vee \psi$ and $\phi \wedge \psi$. Essentially, whenever a negation appears, the polarity of its subterm is reversed. If the polarity is positive, the Boolean value is chosen to "ignore" the predicate in the body of whatever quantifier's variable caused the overflow: true for universally quantified variables and false otherwise. If the polarity is negative, the Boolean value chosen for an overflowing integer predicate should be the negation of the Boolean value chosen when polarity is positive.

Our algorithm for NOBV takes a term and returns a term augmented with guards that overwrite the value of overflowing predicates as described above. We define NOBV recursively over the term being transformed as the function

$$\textsc{fixOverflow}(term : Term, polarity : Bool)$$

where *polarity* is the polarity of the *term* argument with respect to the top-level term.

First we define NOBV for integer operations (Algorithm 1), integer predicates (Algorithm 2, Algorithm 3), and logical connectives (Algorithm 4 and Algorithm 5) to match Milicevic and Jackson. Then, we cover extensions to Milicevic and Jackson's method included in NOBV to cover all of MSFOL.

Algorithm 1 describes how we transform an integer operation ($+, -, /$, etc.) by recursively transforming its arguments. We gather overflow checks bottom up. Overflow checks are collections of predicates that will evaluate to true if the term overflows. We collect these overflow checks in two sets: one for terms containing universally quantified variables and one for the rest, which we refer to as existentially quantified. There are no polarity considerations in Algorithm 1 because this algorithm is only for integer operators (not predicates).

Algorithm 1 FIXOVERFLOW Where f is an Integer Operation under NOBV

function FIXOVERFLOW$(f(args), polarity)$
  for $i \leftarrow 0 \ldots |args| - 1$ **do**
    $(args'[i], overflows_i^{\forall}, overflows_i^{\exists}) \leftarrow$ FIXOVERFLOW$(args[i], polarity)$
  end for
  $\phi \leftarrow f(args')$
  $overflows^{\forall} \leftarrow \bigcup_{0 \leq i < |args|} overflows_i^{\forall}$
  $overflows^{\exists} \leftarrow \bigcup_{0 \leq i < |args|} overflows_i^{\exists}$
  if CONTAINSUNIVVAR(ϕ) **then** ▷ Contains free $\forall$ quantified var
    $overflows^{\forall} \leftarrow overflows^{\forall} \cup \{$OVERFLOWS$(\phi)\}$
  else
    $overflows^{\exists} \leftarrow overflows^{\exists} \cup \{$OVERFLOWS$(\phi)\}$
  end if
  return $(\phi, overflows^{\forall}, overflows^{\exists})$
end function

Algorithm 2 describes how, given an integer predicate $p(args)$, NOBV transforms it so that it evaluates to true or false as in Milicevic and Jackson's method. The overflow checks collected from recursive calls on the arguments of the integer predicate are passed to Algorithm 3, described later. Again, polarity remains unchanged as integer predicates $(=, >, <,$ etc.) do not affect polarity.

Algorithm 2 FIXOVERFLOW for Integer Predicate p under NOBV

function FIXOVERFLOW$(p(args), polarity)$
  for $i \leftarrow 0 \ldots |args| - 1$ **do**
    $(args'[i], overflows_i^{\forall}, overflows_i^{\exists}) \leftarrow$ FIXOVERFLOW$(args[i], polarity)$
  end for
  $overflows^{\forall} \leftarrow \bigcup_{0 \leq i < |args|} overflows_i^{\forall}$
  $overflows^{\exists} \leftarrow \bigcup_{0 \leq i < |args|} overflows_i^{\exists}$
  $\phi \leftarrow$ OVERFLOWGUARD$(p(args'), overflows^{\forall}, overflows^{\exists}, polarity)$ ▷ Algorithm 3
  return $(\phi, overflows^{\forall}, overflows^{\exists})$
end function

Algorithm 3 describes the process of transforming a predicate so that if any of its subterms overflow, the potentially incorrect value to which the predicate evaluates is ignored

in favor of the chosen Boolean value. We use

$$\textsc{disjunction}(A) \equiv \begin{cases} \bot & \text{if } A = \emptyset \\ \bigvee_{a \in A} a & \text{otherwise} \end{cases} \tag{3.5}$$

to combine the sets of overflow checks into a single term that is true if and only if a subterm of the statement overflows existentially or universally. The polarity of an integer predicate is used to decide how the universal and existential overflow checks should be arranged to make the new term evaluate to true or false if the predicate would overflow.

Algorithm 3 Guarding a Predicate when Overflows Occur

```
1:  function OVERFLOWGUARD(φ, overflows^∀, overflows^∃, polarity)
2:      checks^∀ ← DISJUNCTION(overflows^∀)          ▷ Equation 3.5.
3:      checks^∃ ← DISJUNCTION(overflows^∃)
4:      if polarity then
5:          φ′ ← (φ ∨ checks^∀) ∧ ¬checks^∃
6:      else
7:          φ′ ← (φ ∨ checks^∃) ∧ ¬checks^∀
8:      end if
9:      return φ′
10: end function
```

One difference in presentation between Milicevic and Jackson's method and NOBV is that Milicevic and Jackson combine one set of overflow checks as a conjunction of the terms not overflowing and the other as a disjunction of the terms overflowing. For simplicity when supporting additional language constructs (discussed later), we represent the former as the negation of the disjunction of the terms overflowing on lines 5 and 7 of Algorithm 3.

For the sake of brevity, we do not describe FIXOVERFLOW's exact semantics over every logical connective. For any sort of Boolean operand ($\wedge, \vee, \Rightarrow, \neg$, etc.) we set polarity appropriately before calling FIXOVERFLOW on the argument subterms and recombining the results. As an example, Algorithm 4 describes how FIXOVERFLOW inverts the polarity for subterms of applications of $\neg$. Algorithm 5 describes how $\wedge$ does not change polarity when recursing to its subterms. Milicevic and Jackson do not change polarity when recurring to its subterms for all of the binary logical connectives ($\wedge$, $\vee$, $\Leftrightarrow$, and $\Rightarrow$). We differ in how polarity is considered in subterms of $\Leftrightarrow$ and $\Rightarrow$ (see Section 3.1.1), but the process is otherwise unchanged.

Algorithm 4 FIXOVERFLOW for $\neg$ in NOBV and OPFI

function FIXOVERFLOW($\neg a, polarity$)
  $polarity' \leftarrow \neg polarity$
  $(a', overflows^{\forall}, overflows^{\exists}) \leftarrow$ FIXOVERFLOW$(a, polarity')$
  $\phi \leftarrow \neg a'$
  return $(\phi, overflows^{\forall}, overflows^{\exists})$
end function

Algorithm 5 FIXOVERFLOW for $\wedge$ in NOBV and OPFI

function FIXOVERFLOW($a \wedge b, polarity$)
  $(a', overflows_a^{\forall}, overflows_a^{\exists}) \leftarrow$ FIXOVERFLOW$(a, polarity)$
  $(b', overflows_b^{\forall}, overflows_b^{\exists}) \leftarrow$ FIXOVERFLOW$(b, polarity)$
  $overflows^{\forall} \leftarrow overflows_a^{\forall} \cup overflows_b^{\forall}$
  $overflows^{\exists} \leftarrow overflows_a^{\exists} \cup overflows_b^{\exists}$
  $\phi \leftarrow a' \wedge b'$
  return $(\phi, overflows^{\forall}, overflows^{\exists})$
end function

3.1.1 Extensions in NOBV

Fortress operates on multi-sorted first-order logic (MSFOL) problems, which it passes to an SMT solver. Milicevic and Jackson's presentation (for the Alloy language) assumes language constructs such as uninterpreted functions and quantification have already been reduced to propositional logic. And, thus, these are not included in Milicevic and Jackson's presentation, which only describes evaluation for integer operations, integer predicates, and logical predicates.

We generalize Milicevic and Jackson's method to support these language features in NOBV. Additionally, Milicevic and Jackson define the polarity of the subterms of applications of $\Rightarrow$ and $\Leftrightarrow$ to be the same as the polarity of the terms themselves. We provide an alternative definition of polarity for these subterms. Finally, in *NOBV* we support if-then-else over terms of arbitrary sorts (which is not supported by Alloy) and interpreted functions (function definitions), both of which are not handled in Milicevic and Jackson's method.

Uninterpreted Functions and Quantification

Milicevic and Jackson's method applies only to a predefined set of integer operations that take only integers as arguments. In MSFOL, uninterpreted functions can accept arbitrary sorts for arguments and result types. A formula may be an input to an uninterpreted function in NOBV.

In NOBV, to support all of MSFOL, a Boolean argument may contain a quantifier. Subterms of this quantifier that contain the quantified term cannot be included in any checks outside the scope of the quantifier. For example, with $f : Bool \times Int \to Int$ and $p : Int \to Bool$,

$$\forall x \bullet f\,(\exists y \bullet 0 \leq x \wedge p(y+5), x+1) = x$$

the check for $y{+}5$ overflowing can be applied to the predicate $p(y{+}5)$, but not to $f(\dots) = x$.

Algorithm 6 describes how we exclude any terms that contain variables that are not quantified at the level of the predicate. These terms are still used for the predicates within the quantifier's body, so the quantifier will now have the expected value while "skipping" values that cause an overflow. Predicates with this quantifier as a subterm will still check subterms that do not contain these quantified variables and so will still detect overflows caused by the quantifiers so far.

Algorithm 6 FIXOVERFLOW for Quantifiers in NOBV and OPFI

Require: $\mathcal{Q}$ is $\forall$ or $\exists$
  function FIXOVERFLOW($\mathcal{Q}x : A \bullet \phi, polarity$)
    $(\phi', overflows^{\forall}, overflows^{\exists}) \leftarrow$ FIXOVERFLOW($\phi, polarity$)
    $overflows^{\forall} \leftarrow \left\{c \in overflows^{\forall} | x \notin \text{FREEVARS}(c)\right\}$
    $overflows^{\exists} \leftarrow \left\{c \in overflows^{\exists} | x \notin \text{FREEVARS}(c)\right\}$
    return $(\mathcal{Q}x : A \bullet \phi', overflows^{\forall}, overflows^{\exists})$
  end function

Polarity of $\Rightarrow$ and $\Leftrightarrow$

The logical connectives $\Rightarrow$ and $\Leftrightarrow$ can be defined in terms of $\wedge$, $\vee$ and $\neg$ as

$$a \Rightarrow b \equiv \neg a \vee b$$

and

$$a \Leftrightarrow b \equiv (a \wedge b) \vee (\neg a \wedge \neg b)$$

Boolean equality ($a = b$ where $a, b : Bool$) is equivalent to $\Leftrightarrow$, so for the purposes of this discussion of polarity we will refer only to $\Leftrightarrow$. Equality of other sorts is treated as other non-integer functions are treated. When $a = b$ is considered over a non-Boolean, non-integer sort, it is treated as any other function invocation.

Milicevic and Jackson ignore the negations in the definitions of the logical connectives $\Rightarrow$ and $\Leftrightarrow$ when defining polarity, only inverting polarity for subterms of $\neg$, leading to incorrect results. Consider the formula

$$\forall x \bullet (x + 100 < x) \Rightarrow \bot \tag{3.6}$$

For any x, $x + 100 < x$ is false and $false \Rightarrow^{\mathcal{I}} false$ is true. Therefore, this formula is true for unbounded integers. However, when an overflow occurs in $x + 100 < x$, it evaluates to true (as x is universally quantified and the term's polarity is positive). As $true \Rightarrow^{\mathcal{I}} false$ is false, the formula is incorrectly false. Alloy incorrectly fails to find an instance for which Equation 3.6 is true, but for

$$\forall x \bullet \neg(x + 100 < x) \vee \bot$$

which should be equivalent, it correctly finds a satisfying instances. We correct this error by altering the definition of polarity to include that in $a \Rightarrow b$, a has the opposite polarity of $a \Rightarrow b$.

Milicevic and Jackson's method also incorrectly handles the polarity of subterms of the $\Leftrightarrow$ logical connective. Consider the formula

$$\forall x : Int \bullet \phi \Leftrightarrow \psi \tag{3.7}$$

where ϕ and ψ are arbitrary integer predicates in terms of x. In Alloy, the polarity of ϕ (and ψ) is incorrectly considered to be the same polarity as $\phi \Leftrightarrow \psi$. In our correctness evaluation, we demonstrate this incorrect behavior on the formula

$$\forall x, y \bullet (x - y > 0) \Leftrightarrow x > y$$

If ϕ overflows, $\phi \Leftrightarrow \psi$ should be true so the chosen x is "skipped" in the quantifier. In order to make $\phi \Leftrightarrow \psi$ true when ϕ overflows, ϕ must be equal to ψ. As we must statically transform the problem before solving, rather than dynamically choose values during evaluation, $\phi = \psi$ is insufficient; we must set ϕ to either true or false. However, we cannot determine the value of ψ before solving, so we do not know the value ϕ should be if an overflow occurs. If we instead transform Equation 3.7 to

$$\forall x : Int \bullet (\phi \wedge \psi) \vee (\neg\phi \wedge \neg\psi)$$

we can evaluate ϕ (and ψ) at both negative and positive polarity. Algorithm 7 describes how we perform this transformation in place when recursing through subterms.

Algorithm 7 FIXOVERFLOW for $\Leftrightarrow$ (and Boolean Equality) in NOBV and OPFI

function FIXOVERFLOW($a \Leftrightarrow b, polarity$)
  $\phi \leftarrow (a \wedge b) \vee (\neg a \wedge \neg b)$
  return FIXOVERFLOW($\phi, polarity$)
end function

If-Then-Else

Alloy supports if-then-else (ITE) statements (written `c implies t else f`) only when the branches are of sort *Bool*. Milicevic and Jackson's method does not mention this language construct, presumably reducing it to propositional logic beforehand. Fortress supports if-then-else statements where the branches are of an arbitrary sort. NOBV supports ITE to avoid having to hoist the ITE construct to a predicate, which is often called if-lifting [19]. Consider abstractly $\text{ITE}(c, t, f)$ where c, t, and f are arbitrary terms. Ignoring integers for a moment, semantically

$$\text{ITE}(c, t, f) = \begin{cases} t \text{ if } c \\ f \text{ if } \neg c \end{cases}$$

which can be simplified to $(c \wedge t) \vee (\neg c \wedge f)$ if t and f are Boolean. However, if c contains an integer overflow, both c and $\neg c$ will have the same value. For illustration, consider the case where t and f are Boolean. If c and $\neg c$ are both true, this simplification will act as $t \vee f$. If c and $\neg c$ are both false, it will instead be false. Therefore, if c overflows, this simplification is incorrect. Consider the formula

$$\forall x \bullet \text{ITE}(x + 100 < x, \bot, \top)$$

which is false for unbounded integers. Under the bounded integer range $[-16, 15]$, c always overflows. If we use the $(c \wedge t) \vee (\neg c \wedge f)$ simplification, the body of the quantifier would be equivalent to $\bot \vee \top$ for every value (because $x + 100 < x$ and its negation are both true), making the entire formula incorrectly evaluate to true. If t and f are non-Boolean terms, overflows within them still must be considered if and only if c indicates their branch should be taken and c does not overflow.

Instead, we use Algorithm 8, which describes how we construct the overflow checks for the ITE construct to ensure the checks for the t and f subterms are only evaluated if c does not overflow *and* the respective branch is taken. The entire term is said to overflow existentially if c overflows existentially or c does not overflow (universally *or* existentially)

and the chosen branch overflows existentially (c is true and t overflows existentially or c is false and f overflows existentially). The same is true for the term overflowing universally.

Algorithm 8 ITE Overflow Prevention

function FIXOVERFLOW(ITE$(c, t, f), polarity$)
  $(c', overflows_c^\forall, overflows_c^\exists) \leftarrow$ FIXOVERFLOW$(c, polarity)$
  $(t', overflows_t^\forall, overflows_t^\exists) \leftarrow$ FIXOVERFLOW$(t, polarity)$
  $(f', overflows_f^\forall, overflows_f^\exists) \leftarrow$ FIXOVERFLOW$(f, polarity)$
  $checks_c \leftarrow$ DISJUNCTION$(overflows_c^\exists \cup overflows_c^\forall)$ ▷ True iff c overflows
  $checks_t^\forall \leftarrow c \wedge$ DISJUNCTION$(overflows_t^\forall)$ ▷ t is taken and overflows universally
  $checks_t^\exists \leftarrow c \wedge$ DISJUNCTION$(overflows_t^\exists)$ ▷ t is taken and overflows existentially
  $checks_f^\forall \leftarrow \neg c \wedge$ DISJUNCTION$(overflows_f^\forall)$ ▷ f is taken and overflows universally
  $checks_f^\exists \leftarrow \neg c \wedge$ DISJUNCTION$(overflows_f^\exists)$ ▷ f is taken and overflows existentially
  $overflows^\forall \leftarrow overflows_c^\forall \cup \{\text{ITE}(\neg checks_c, checks_t^\forall, checks_f^\forall)\}$
  $overflows^\exists \leftarrow overflows_c^\exists \cup \{\text{ITE}(\neg checks_c, checks_t^\exists, checks_f^\exists)\}$
  $\phi \leftarrow \text{ITE}(c', t', f')$
  if The type of t is *Bool* **then**
    $\phi \leftarrow$ OVERFLOWGUARD$(\phi, overflows^\forall, overflows^\exists, polarity)$
  end if
  return $(\phi, overflows^\forall, overflows^\exists)$
end function

Interpreted Function Definitions

Fortress supports user-defined (interpreted) functions. The definition of an interpreted function $f : t_1 \times \ldots t_n \rightarrow t_{body}$ is a sequence of n parameters $p_1 \ldots p_n$ and a body term, $\phi : t_{body}$, which may contain integer operations. These integer operations have the potential to overflow. A definition of a function is written $f(p_1, \ldots p_n) := \phi$.

Algorithm 9 describes how, given a function definition $f(p_1, \ldots p_n) := \phi$ of type $f : t_1 \times \ldots t_n \rightarrow t_{body}$, where ϕ contains no integer predicates, we modify its invocation $f(a_1, \ldots a_n)$. When determining the subterms of a predicate for overflow checks, if an interpreted function invocation is a subterm of the predicate, we add additional checks to represent terms in the body of the function overflowing. We gather all the overflowable subterms of the body in terms of the parameters. Then, we substitute the arguments for the parameters in the overflowable terms. If the function is an integer predicate (its result sort is *Bool* and it takes an integer as an argument), then we apply guards as with any other integer predicate.

Algorithm 9 Overflow Prevention for Invocation of Interpreted Function $f(args)$ where $f(params) = \beta$ in NOBV

Require: β contains no integer predicates
  function FIXOVERFLOW($f(args), polarity$)
    for $i \leftarrow 0 \ldots |args| - 1$ **do**
      $(args'_i, overflows_i^{\forall}, overflows_i^{\exists}) \leftarrow$ FIXOVERFLOW($args[i], polarity$)
    end for
    $overflows_{args}^{\forall} \leftarrow \bigcup_i overflows_i^{\forall}$
    $overflows_{args}^{\exists} \leftarrow \bigcup_i overflows_i^{\exists}$
    ▷ This can be precomputed.
    $(\_, overflows_{\beta}^{\forall}, overflows_{\beta}^{\exists}) \leftarrow$ FIXOVERFLOW($\beta, polarity$)
    ▷ Substitute args for params
    $overflows_{\beta} \leftarrow \left\{\phi\,[params|args'] \middle| \phi \in overflows_{\beta}^{\forall} \cup overflows_{\beta}^{\exists}\right\}$
    $overflows_{\beta}^{\forall} \leftarrow \{\phi | \text{CONTAINSUNIVVAR}(\phi)\}$ ▷ Repartition $overflows_{\beta}$
    $overflows_{\beta}^{\exists} \leftarrow overflows_{\beta} - overflows_{\beta}^{\forall}$
    $overflows^{\forall} \leftarrow overflows_{args}^{\forall} \cup overflows_{\beta}^{\forall}$
    $overflows^{\exists} \leftarrow overflows_{args}^{\exists} \cup overflows_{\beta}^{\exists}$
    $\phi \leftarrow f(args')$
    if f is a predicate **then**
      $\phi \leftarrow$ OVERFLOWGUARD($\phi, overflows^{\forall}, overflows^{\exists}, polarity$)
    end if
    return $(\phi, overflows^{\forall}, overflows^{\exists})$
  end function

If the body of a function definition contains an integer predicate, we must apply guards to determine the value that the predicate should be if an overflow occurs. However, as an argument could be universally or existentially quantified, we cannot statically determine if a parameter should be treated existentially or universally. Thus, for an interpreted function that has a body that contains an integer predicate, we cannot use the above method. Instead, we axiomatize all the function definitions that contain integer predicates before using FIXOVERFLOW on any formulas. When axiomatized, a function definition $f(p_1, \ldots p_n) := \phi$ becomes $\forall p_1 \ldots p_n \bullet f(p_1, \ldots p_n) = \phi$.

When NOBV is implemented as a transformer in Fortress, it is applied directly after the transformation of integers to bitvectors.

3.2 Overflow Preventing Finite Integers (OPFI)

When expressions overflow in NOBV, certain expressions can be evaluated incorrectly with regards to unbounded integers. Consider the obviously false formula

$$\forall x \bullet x + 100 - 100 \neq x \tag{3.8}$$

over the integer range $[-16, 15]$. In NOBV, every possible value of the universally quantified variable x within range will cause the expression $x{+}100$ to overflow, forcing $x{+}100{-}100 \neq x$ to evaluate to true, making the entire formula evaluate to true. Similarly,

$$\exists x \bullet x + 100 - 100 = x \tag{3.9}$$

should evaluate to true, but in NOBV overflows cause the formula to incorrectly evaluate to false. In both cases, if even a single value for x does not cause an overflow, the formulas will evaluate correctly. This incorrect behavior is observable in more complicated problems. Consider

$$\exists x_1 \ldots x_{i+1} \bullet \left(\bigwedge_{1 \leq k \leq i+1} x_k > 1 \right) \wedge \frac{(x_1 + \cdots + x_{i+1})}{i} > 1 \tag{3.10}$$

where i is the maximum value in the finite range of integers. Simplification cannot be used to obtain the correct result. In unbounded integers, the assignment where every $x_k = 1$ is in range would make the formula evaluate to true. However, any assignment that would make $(x_1 + \cdots + x_{i+1})/i > 1$ true will cause an overflow when using bitvectors. We checked these problems in the Alloy Analyzer and got the expected incorrect results.

When no overflows occur, a formula in NOBV will have the same value as it would for unbounded integers, and thus be correct. Therefore, a similar approach that overflows

in fewer cases will get incorrect results in fewer cases as well. SAT solvers must always represent values finitely, such as using bitvectors to represent integers. Unfortunately, by the nature of bitvectors, if *any* subterm is out of range, an overflow occurs and the result is unknown. However, SMT solvers can represent integers symbolically, avoiding this limitation of bitvectors. If a finite range for integers is used wherever a solver needs to search through values, the problem is decidable. By using unbounded integers except for the search range, we can reduce how often overflows occur.

Based on this insight, we present our novel method for handling integer overflow, called overflow preventing finite integers (OPFI). We begin by finitizing the problem using a new finite sort $\mathbb{I}$ to represent integers wherever the solver must search for values, much in the same way that we convert the problem to fixed-width bitvectors when using NOBV. Then, we guard predicates with overflow checks to avoid any incorrect behavior our finitization introduced. However, we do *not* need to check for overflows of integer operations. In OPFI, bit vectors are not used. Rather, the original formula is transformed to one that uses unbounded integers within value operations and a new, finite sort for the range of integer values for quantified variables.

In order to have a finite representation of integers in the search space, we create a finite sort $\mathbb{I}$ with scope size equal to the number of values in our finite range $r = [MIN, MAX]$. We denote the value in our new sort that represents the integer x as v_x.

In order to use SMT solvers' infinite representation of integers when performing operations, we then construct two total interpreted functions to cast values between $\mathbb{I}$ and unbounded integers. We define $\textsc{toInt} : \mathbb{I} \to Int$ as

$$\textsc{toInt}(v_x) := x$$

We define the reverse similarly: $\textsc{fromInt} : Int \to \mathbb{I}$ is

$$\textsc{fromInt}(x) = \begin{cases} v_x & \text{if } MIN \leq x \leq MAX \\ v_{MIN} & \text{otherwise} \end{cases}$$

where we use v_{MIN} as a default so that fromInt is a total function, although any value within the range $[MIN, MAX]$ can be used. Whenever fromInt is called on a value out of range, we lose information in the same way as an overflow in a bitvector operation. NOBV detects the loss of information from bitvectors (so that it can "ignore" the current variable assignments) by checking each term that can overflow. We use this method as the basis for OPFI, only instead of every bitvector operation needing to be checked for overflow, we only need to check for values out of range when we coerce unbounded integers back to our finite sort. We are able to leave interpreted functions over unbounded integers.

In order to finitize the problem, the search space of the MSFMF problem we give to the SMT solver must be finite. The two places unbounded integers would prevent this from happening is quantifiers (including the implicit existential quantification of constants) and uninterpreted functions (both as parameters and result sorts). Outside of a decidable fragment of integer logic, a solver cannot check that a formula is true for every possible value of a universally quantified variable. Nor can a solver finitely enumerate every possible value to determine if a formula is true for a single possible value of an existentially quantified variable. Similarly, the MSFMF problem implicitly existentially quantifies uninterpreted functions, searching for some interpretation where the set of formulas given is satisfiable. If an uninterpreted function has an argument of sort Int, or if an uninterpreted function has a result sort of Int, then that function has infinitely many possible value assignments, potentially making any MSFMF problem that contains it undecidable.[2]

[2]If an uninterpreted function has a result sort with scope size one, its values are finitely enumerable as all inputs are mapped to the single possible output value.

Algorithm 10 Recursive Elimination of Int in OPFI

```
function REPLACEINT(φ)
    match φ do
        case v where v is a variable
            return v
        case Qx : A • ψ where Q is ∀ or ∃
            if A = Int then                          ▷ We will quantify over 𝕀 instead
                ψ' ← ψ [x|TOINT(x)]                  ▷ x will be 𝕀, so we cast x to Int for arithmetic
                ψ' ←REPLACEINT(ψ')
                return Qx : 𝕀 • ψ'                   ▷ Quantify over 𝕀
            else
                ψ' ←REPLACEINT(ψ)
                return Qx : A • ψ'
            end if
        case f(a_1, ... a_n) where f : A_1 × ... × A_n : B    ▷ Including 0-arity constants
            for x ∈ [1, n] do
                a_x ← REPLACEINT(a_x)
                if A_x = Int and f is uninterpreted then      ▷ Cast args if needed
                    a_x ← FROMINT(a_x)
                end if
            end for
            if B = Int then                          ▷ We will replace Int with 𝕀
                return TOINT(f(a_1, ... a_n))        ▷ So, we must cast back to Int
            else
                return f(a_1, ... a_n)
            end if
end function
```

To ensure we finitize the search space of the problem, we replace the Int sort with $\mathbb{I}$ in quantifiers and uninterpreted functions. Algorithm 10 explains how, given a (universally or existentially) quantified term of the form $\exists v : Int \bullet \phi$, we replace it with $\exists v : \mathbb{I} \bullet \phi\,[v|\textsc{toInt}(v)]$. For example, $\forall x : Int \bullet x + 1 < 0$ becomes $\forall x : \mathbb{I} \bullet \textsc{toInt}(x) + 1 < 0$.

For uninterpreted functions, if any parameters are Int, we replace Int with $\mathbb{I}$ and cast its arguments to $\mathbb{I}$ using FROMINT any time the function is invoked. Similarly, if the result sort of an uninterpreted function is Int we replace it with $\mathbb{I}$ to ensure the search space is finite. We then must cast the result of these invocations back to Int to continue operating on the result. The case for $f(a_1, \ldots, a_n)$ in Algorithm 10 covers all the interpreted functions

also. If a function is interpreted, the transformation recurses into the arguments, but we do not change the sorts of the parameters or the result. So, toInt and fromInt are not used.

Algorithm 11 Finitization of Integers in OPFI

Require: $\mathcal{P} = (\Sigma, \Gamma, \mathcal{S})$ is an MSFOL problem
Require: $\Sigma = (\Theta, \mathcal{F})$
  Let $\mathbb{I}$ be a new sort symbol.
  $\Theta \leftarrow \Theta \cup \{\mathbb{I}\}$ ▷ Add $\mathbb{I}$ to the set of sorts in $\mathcal{P}$
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathbb{I} \rightarrow |[MIN, MAX]|\}$ ▷ Set the scope for $\mathbb{I}$
  $\Gamma' \leftarrow \emptyset$ ▷ Replace Int with $\mathbb{I}$ in Γ
  for $\gamma \in \Gamma$ **do**
    $\Gamma' \leftarrow \Gamma' \cup \{\text{replaceInt}(\gamma)\}$ ▷ Algorithm 10
  end for
  $\Gamma \leftarrow \Gamma'$
  $\mathcal{F}' \leftarrow \emptyset$ ▷ Replace Int with $\mathbb{I}$ in uninterpreted functions
  for $f : A_1 \times \ldots \times A_n : R \in \mathcal{F}$ **do**
    if f is an uninterpreted function **then**
      Replace any occurrences of Int in $A_1 \ldots A_n, R$ with $\mathbb{I}$ in f
      $\mathcal{F}' \leftarrow \mathcal{F}' \cup \{f\}$
    end if
  end for
  $\mathcal{F} \leftarrow \mathcal{F}'$
  $\mathcal{F} \leftarrow \mathcal{F} \cup \{\text{toInt}, \text{fromInt}\}$

Algorithm 11 describes how, given an MSFMF problem $\mathcal{P}$ containing integers, we finitize the use of integer for a given range $[MIN, MAX]$. We introduce $\mathbb{I}$, use Algorithm 10 to eliminate integer uses in $\mathcal{P}$'s formulas, and replace Int with $\mathbb{I}$ for uninterpreted functions in $\mathcal{P}$'s signature. We then add fromInt and toInt to the signature of the problem.

As mentioned previously, whenever fromInt is invoked, if the argument is out of range we lose information in an "overflow". Algorithm 12 describes how we check for overflows in the arguments to fromInt by gathering checks to later use in applying guards to predicates much in the same way as we do for NOBV.

For an integer predicate in OPFI, the values of its arguments are unbounded integers rather than just the values in range. So, we check the arguments to integer predicates for overflows. We could cast the arguments to integer predicates to $\mathbb{I}$ using fromInt and

Algorithm 12 Checking FROMINT for Overflow in OPFI

function FIXOVERFLOW(FROMINT(ϕ), $polarity$)
  $(\phi', overflows^{\forall}, overflows^{\exists}) \leftarrow$ FIXOVERFLOW($\phi, polarity$)
  $overflowCheck = \phi' < MIN \vee \phi' > MAX$ ▷ True iff ϕ' is out of range
  if CONTAINSUNIVVAR(ϕ') **then**
    $overflows^{\forall} \leftarrow overflows^{\forall} \cup \{overflowCheck\}$
  else
    $overflows^{\exists} \leftarrow overflows^{\exists} \cup \{overflowCheck\}$
  end if
  return (FROMINT(ϕ'), $overflows^{\forall}, overflows^{\exists}$)
end function

only check arguments of FROMINT , but this would require writing definitions for integer predicates over elements of set $\mathbb{I}$. As we use unbounded integers to represent values in value computations, we no longer need to check if integer operations overflow as in Algorithm 1. Instead, Algorithm 13 describes how we alter Algorithm 2 to include overflow checks for arguments to integer *predicates*. We no longer check for overflows in integer operations, so Algorithm 1 is not used. Instead we simply recurse into the arguments of integer operations without generating overflow checks.

Algorithm 13 FIXOVERFLOW for Integer Predicate p under OPFI

function FIXOVERFLOW($p(args), polarity$)
  for $i \leftarrow 0 \ldots |args| - 1$ **do**
    $(args'[i], overflows_i^{\forall}, overflows_i^{\exists}) \leftarrow$ FIXOVERFLOW($args[i], polarity$)
    $overflowCheck \leftarrow args'[i] < MIN \vee args'[i] > MAX$ ▷ Alter Algorithm 2
    if CONTAINSUNIVVAR($args'[i]$) **then**
      $overflows_i^{\forall} \leftarrow overflows_i^{\forall} \cup \{overflowCheck\}$
    else
      $overflows_i^{\exists} \leftarrow overflows_i^{\exists} \cup \{overflowCheck\}$
    end if ▷ End alterations of Algorithm 2
  end for
  $overflows^{\forall} \leftarrow \bigcup_{1 \leq i < |args|} overflows_i^{\forall}$
  $overflows^{\exists} \leftarrow \bigcup_{1 \leq i < |args|} overflows_i^{\exists}$
  $\phi \leftarrow$ OVERFLOWGUARD($p(args'), overflows^{\forall}, overflows^{\exists}, polarity$)
  return ($\phi, overflows^{\forall}, overflows^{\exists}$)
end function

Table 3.1 lists the algorithms discussed in this chapter by approach for easier reference.

Table 3.1: Algorithms by Approach

Case considered	NOBV	OPFI
Integer operation $f(args)$	Algorithm 1	–
Integer predicate $p(args)$	Algorithm 2	Algorithm 13
Guarding predicate from overflow	Algorithm 3	
$\neg a$	Algorithm 4	
$a \wedge b$	Algorithm 5	
Quantifiers $\mathcal{Q}x \bullet \phi$	Algorithm 6	
$a \Leftrightarrow b$	Algorithm 7	
ITE(c, t, f)	Algorithm 8	
Interpreted function $f(args)$	Algorithm 9	–*
Recursive elimination of Int	–	Algorithm 10
Finitization of integers	–	Algorithm 11
FROMINT(a)	–	Algorithm 12

* OPFI axiomatizes interpreted functions containing integer predicates.

By continuing to use unbounded integers for integer operations, we are able to gain several advantages over NOBV.

1. NOBV requires checking for an overflow for every integer operation, while OPFI only checks the results of the top-most integer operations that are arguments to predicates and uninterpreted functions.

2. The overflow checks are simpler to write. For a given expression ϕ we can check if it overflows with the term $\phi < MIN \vee \phi > MAX$. Compared to NOBV, there are fewer terms in the problem we send to the solver, and it is simpler to implement than static overflow checks for fixed-width bitvectors.

3. Bitvectors are never used for integers in OPFI. Together with the quantifier expansion step in the Fortress method for MSFMF, the problem contains EUF and unbounded integers for values. Additionally, constants outside the finite range remain representable.

4. By not relying on bitvectors to represent integers, we are not constrained to a range of the form $[-n, n-1]$. Thus, if the problem uses only about natural numbers, we

can effectively halve the search space by instead using the range $[0, n-1]$ or keep the search space the same but search twice as many useful values with the range $[0, 2n-1]$.

3.3 Evaluation

We evaluate the unchecked bitvector (UBV), NOBV, and OPFI methods implemented in Fortress for correctness and performance. For correctness evaluation, we constructed a set of problems and compared each result to that of the same problem using unbounded integers. The correctness evaluation consists of problems specifically chosen to highlight the differences between methods. This set is not intended to be indicative of typical problems. For performance evaluation, we compared the performance of the methods on problems in the SMT-LIB UFNIA (uninterpreted functions and non-integer arithmetic) benchmark set [3].

3.3.1 Correctness Evaluation

Table 3.2: Integer Methods Correctness Evaluation

	Correct	Alloy	Unbounded	OPFI	NOBV	UBV
$\forall x \bullet x + 100 - 100 \neq x$	UNSAT	SAT	UNSAT	UNSAT	UNSAT	UNSAT
$\exists x \bullet x + 100 - 100 = x$	SAT	UNSAT	SAT	SAT	SAT	SAT
$\bigvee_{i=1}^{4} x_i > 1 \wedge (\sum_{i=1}^{4} x_i)/3 > 1$	SAT	UNSAT	SAT	SAT	UNSAT	UNSAT
$\forall x, y \bullet (x - y > 0) \Leftrightarrow (x > y)$	SAT	UNSAT	SAT	SAT	SAT	UNSAT
$\forall x \bullet (x + 100 < x) \Rightarrow \bot$	SAT	UNSAT	SAT	SAT	SAT	UNSAT
$\forall x \bullet \neg(x + 100 < x) \vee \bot$	SAT	SAT	SAT	SAT	SAT	UNSAT
$\forall x \bullet \mathrm{ITE}(x + 100 < x, \bot, \top)$	SAT	UNSAT	SAT	SAT	SAT	UNSAT
$\forall x \exists y \bullet x + 1 < y$	SAT	SAT	SAT	SAT	SAT	SAT
$\forall x \bullet f(x) = \mathrm{ITE}(x > 0, x^2, -x)$	SAT	SAT	TO	SAT	SAT	UNSAT
$\forall x \bullet x + 100 < x$	UNSAT	SAT	UNSAT	UNSAT	UNSAT	UNSAT
$\forall x \bullet f(x) = 2x \wedge g(x) = x/2 \wedge g(f(x)) = x$	SAT	UNSAT	SAT	UNSAT	UNSAT	UNSAT

TO represents a timeout after 20 minutes

We review the correctness of various methods on 11 problems described in Table 3.2. The evaluation compares the methods' returned SAT or UNSAT result with regards to the correct result over unbounded integers for the purpose of providing examples to discuss the improvements made in this thesis. Table 3.2 presents the results of our correctness tests. Each finite method returns SAT or UNSAT for each problem in under a minute.

The problem

$$\bigvee_{i=1}^{4} x_i > 1 \wedge \frac{\sum_{i=1}^{4} x_i}{3} > 1 \tag{3.11}$$

was evaluated over the finite integer range $[-4, 3]$. The rest of the problems were evaluated over the finite integer range $[-8, 7]$. The problem $\forall x \bullet f(x) = \text{ITE}(x > 0, x^2, -x)$ caused the unbounded integers method to timeout. This problem demonstrates the effectiveness of finite methods because it could not be solved for unbounded integers.

OPFI and NOBV are able to correctly solve several problems that Alloy and the UBV method fail to solve correctly. Equation 3.11 demonstrates that there are problems for which OPFI obtains the correct result and NOBV obtains an incorrect result. The first two problems, which both use $x + 100 - 100$ evaluate correctly for NOBV and UBV, which was unexpected. We believe this result is due to Z3 simplifying the expression before evaluation. Notably, the problem

$$\forall x \bullet f(x) = 2x \wedge g(x) = \frac{x}{2} \wedge g(f(x)) = x$$

caused OPFI and NOBV to both give an incorrect result. So, while OPFI and NOBV correctly solve a number of problems that Milicevic and Jackson's method, implemented in Alloy, fails to correctly solve, they do not completely solve the problem of overflows causing incorrect results.

3.3.2 Performance Evaluation

We evaluate the performance of UBV, NOBV, and OPFI methods on the SMT-LIB UFNIA benchmark set, which consists of problems containing uninterpreted functions and non-linear integer arithmetic in formulas containing quantifiers. We chose this benchmark as linear integer arithmetic is needlessly restrictive for these methods, and we do not support real numbers. Additionally, we include uninterpreted functions, as both NOBV and OPFI support them. We constrain ourselves to files in the benchmark set with only one command to check for satisfiability.

First, we randomize the order of the files in the benchmark set. We run the problem given by each file with a timeout of 30 minutes and a scope size of 16 for all sorts (this is equivalent to a bitwidth of 4 for integers). We record the first 100 problems where all three methods return a result of SAT and the first 100 problems where all three methods return UNSAT without timing out. We run each method three times over these recorded problems and took the mean time for each file as its result time. We evaluate the times

taken for each problem over each method to determine which method(s), if any, solve the problems faster than the others. The performance evaluation is conducted on an Intel® Xeon® CPU E3-1240 v5 @ 3.50GHz x 8 machine running Linux version 4.4.0-210-generic with up to 64GB of user-space memory.

As the times we record are not normally distributed, and we have two independent variables (problem and method) and one dependent variable (running time), we perform a Friedman rank sum test on the running times of the methods across the various problems. The test ranks of each method on every problem (1st, 2nd, or 3rd fastest as 1, 2, or 3) in order to normalize the data. The null hypothesis is that all three of the methods have the same average rank across problems. In these tests, the p-value is the probability that our tests results occurred, supposing the null hypothesis to be true. A small p-value indicates that it is likely that the null hypothesis is false. If a p-value is smaller than a significance level α, then there is statistically significant evidence (at α) to conclude that the null hypothesis is false. We chose $\alpha = 0.05$. Table 3.3 shows that for the SAT and UNSAT problems considered separately and together we can reject the null hypothesis and conclude that at least one method has a different average rank than the others.

Result	p-value
SAT	$< 2.2e{-}16$
UNSAT	$< 2.2e{-}16$
Both	$< 2.2e{-}16$

Table 3.3: Friedman Rank Sum Results for Finite Integer Methods

Since we reject the null hypothesis for the Friedman rank sum test, we use a Pairwise Wilcoxon Rank Sum Test with Bonferroni adjustment as a post-hoc test to compare the ranks of methods pairwise. We do the Friedman test and adjust the Wilcoxon test to be a post-hoc test (rather than simply running the Wilcoxon test pairwise) to avoid compounding errors. The null hypothesis for each pair of methods is that the methods have the same average rank. Table 3.4 shows that there is statistically significant evidence to claim that the NOBV method has a different average rank than OPFI and unchecked bitvectors (UBV). We do not find statistically significance evidence that OPFI performs differently from UBV.

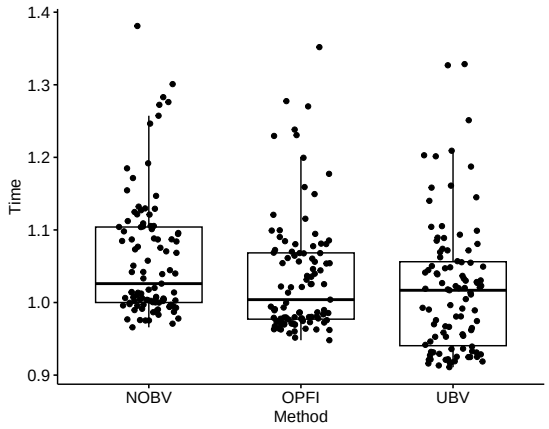
Figure 3.1 shows problems are consistently solved faster when using OPFI than when using NOBV. Figure 3.1c, Figure 3.1a, and Figure 3.1b are box plots of the running times of problems for each of the three methods. OPFI has a faster median time to solve a problem than NOBV, as well as faster first and third quartiles. In Figure 3.1d, every problem is

Result	OPFI vs NOBV	OPFI vs UBV	NOBV vs UBV
SAT	0.00177	0.18530	0.00092
UNSAT	0.041	1.000	0.045
Both	0.0076	0.7086	0.0025

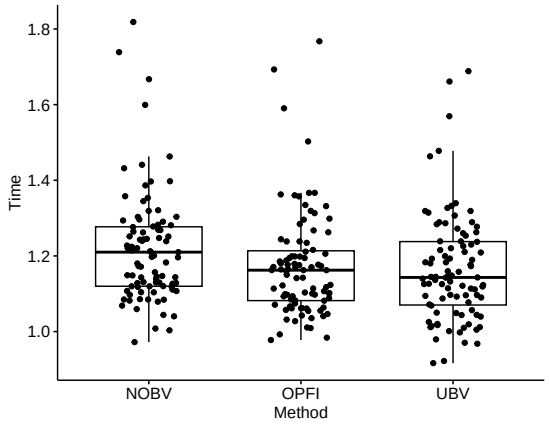
Highlighted values are statistically significant at $\alpha < 0.05$.

Table 3.4: Wilcoxon Post-Hoc p-values for Finite Integer Methods

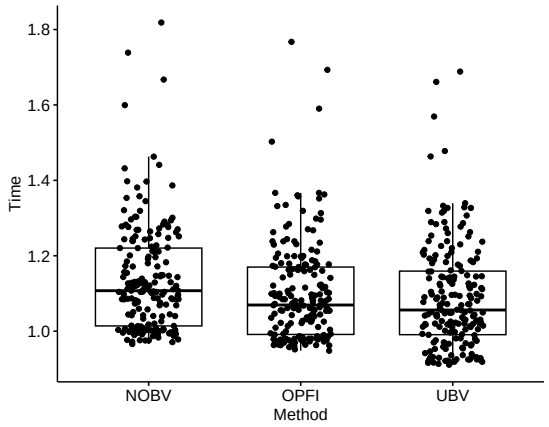
plotted as a point with its x and y coordinates being its running time using OPFI and NOBV respectively. The dashed line is where the running times for the two methods are equal. Most points are above the dashed line, indicating that for the given problem OPFI runs faster. The statistical tests used consider only the ranks of the methods across problems, disregarding the magnitude of differences in running time.



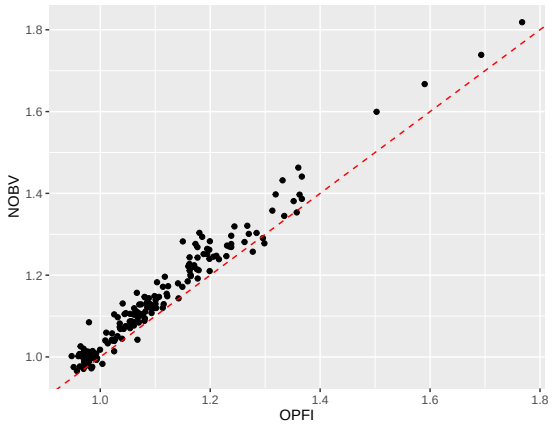
(a) Running Times for SAT problems



(b) Running Times for UNSAT problems



(c) Running Times for both SAT and UNSAT



(d) Running Time of OPFI vs NOBV

These figures omit 1 SAT and 8 UNSAT outliers for legibility.

Figure 3.1: Running Times for Finite Integer Methods

We ran the same 100 SAT and 100 UNSAT problems with unbounded integers. We checked that all methods (including unbounded integers) returned the same SAT or UNSAT result as unbounded integers. Unbounded integers solved the problems faster than the finite methods, which means a future research direction is a hybrid approach using finite and infinite integers within a problem.

3.3.3 Threats to Validity

A potential threat to *internal validity* is that we consider only problems for which all finite methods solve the problem in the 30-minute timeout. One or more methods may perform better on problems for which some other method times out. A possible threat to *external validity* is that the SMT solver can solve all problems included in the study, which could indicate the benchmark set is biased toward including problems solvable using uninterpreted integers, and the finite methods may behave differently on other sets of problems. Another possible threat to *external validity* is that we tested using a single finite integer range. The effectiveness of these methods may change at higher scopes.

3.4 Summary

OPFI is a theoretically more correct integer finitization method than NOBV. On average, OPFI solves problems faster than NOBV within the SMT solver Z3. OPFI introduces minimal performance cost, as it does not solve problems significantly slower that UBV.

Chapter 4

Transitive Closure

Transitive closure is an operation on relations. The transitive closure of a relation is the minimal relation containing every pair in the original relation and additional pairs to make the resulting relation transitive. A relation R is transitive if and only if

$$\forall x, y, z \bullet (x, y) \in R \land (y, z) \in R \Rightarrow (x, z) \in R$$

It is common to use closures in modeling for graph reachability in networks and reasoning about temporal logic. Models in languages like Alloy frequently make use closures. An analysis of over 2000 Alloy models found that over 35% of models used transitive closure operators, and of those that did, the median number of usages per model was 6 [**?**]. This common usage makes it useful that libraries such as Fortress include options for solving transitive closure.

Section 4.1 provides background on the transitive closure operator and the notation we use in this chapter. Section 4.2 discusses several existing methods for finitely representing transitive closure. In Section 4.3, we present our novel representation of transitive closure in FOL for the special case where the problem includes only checking if pair is not a member of the transitive closure. We call this negative transitive closure. Section 4.4 provides a generalization of existing methods to relations of arbitrary arity. Section 4.5 presents a performance evaluation of the methods presented in this chapter.

This chapter presents several contributions:

1. A representation of transitive closure for a finite scope in MSFOL for the special case where the problem only includes checking if a pair is not in the transitive closure (negative transitive closure) (Section 4.3)

2. A generalization of existing MSFOL axiomatizations of transitive closure to include relations of higher arity (Section 4.4)

3. A performance evaluation comparing negative transitive closure to existing axiomatizations (Section 4.5)

4.1 Background

The join operator (;) defined as

$$A; B = \{(x, y) | \exists m : (x, m) \in A \wedge (m, y) \in B\}$$

is used frequently in this chapter. The repeated join of a relation with itself is notated as R^n where n is the number of times R is joined with itself. For example, $R^3 = R; R; R$

Transitive closure is an operator on relations. A relation $R^+ : A \times A$ is the transitive closure of $R : A \times A$ if and only if:

$$R^+ \text{ contains } R : \qquad R \subseteq R^+ \tag{4.1}$$

$$R^+ \text{ is transitive : } \qquad R^+; R^+ \subseteq R^+ \tag{4.2}$$

$$R^+ \text{ is the smallest : } \qquad \forall T \bullet R \subseteq T \wedge T; T \subseteq T \Rightarrow R^+ \subseteq T \tag{4.3}$$

In addition to the normal transitive closure operator, there is also the reflexive transitive closure of R, called R^*, which additionally makes R reflexive.

$$R^* = R^+ \cup \{(x, x) | x \in A\}$$

While Equation 4.1 and Equation 4.2 are expressible in MSFOL, Equation 4.3 is not expressible in MSFOL, as it requires quantifying over a relation. When the sort being closed over has a finite bound, a variety of methods exist to express the transitive closure of a relation via auxiliary functions and axioms in MSFOL.

We use set theory notation to discuss properties of relations, however Fortress and SMT solvers use MSFOL as input and do not support set theory. As such, we must axiomatize any relational properties we wish to express in MSFOL. In set notation, a relation is represented as a set of pairs. To represent the fact that the pair (x, y) is in relation $R : A \times B$, $(x, y) \in R$, using MSFOL, we make R a function $R : A \times B \rightarrow Bool$ and write $R(x, y)$. Similarly, $\neg R(a, b)$ is sometimes written as $(a, b) \notin R$. Using this translation, we refer to auxiliary functions and auxiliary relations interchangeably in this chapter. The auxiliary functions (and relations) are used only to help define the transitive closure of a relation.

4.2 Existing Methods for Expressing Finite Transitive Closure in First-Order Logic

A variety of methods for expressing the transitive closure, R^+, of a relation $R : A \times A$ over a finite sort A of scope n in first-order logic already exist. This section follows the presentation of Tariq [28]. These methods differ in the number of axioms and auxiliary functions used to define the transitive closure of a relation. These methods also differ in the depth of quantifiers they use, which is important for solving MSFMF problems in SMT solvers, as increasing the depth of quantifiers can lead to an explosion in term size when expanding quantifiers.

4.2.1 Simple Iterative

A simple encoding of R^+ is

$$R^+ = R \cup R^2 \cup R^3 \cup \dots \tag{4.4}$$

In this encoding, each join of R with itself will contain all sets of pairs connected by a path of length equal to the number of times the relation is joined with itself. For example if $(x, y) \in R^5$ then there is a path of exactly length 5 from x to y in R. The union of all of these joins is all pairs for which a path exists between two points in R; i.e., the transitive closure of R. The closure of a function over a sort with a scope n can be represented in n steps [17]:

$$R^+ = R \cup (R; R) \cup (R; R; R) \cup R^4 \dots R^n \tag{4.5}$$

where R^k represents all pairs connected by a path of length k, and no more than n edges are needed to connect all pairs of points.

To turn the problem into EUF in Fortress, we expand universal quantifiers so $\forall x : A \bullet \phi$ is equivalent to

$$\bigwedge_{y \in A} \phi\,[x|y]$$

When quantifiers are expanded, the number of terms in a formula grows exponentially with regard to the number of nested quantifiers. In order to avoid an explosion in term size from nested existential quantifiers when representing the intermediate values in every join,

Equation 4.5 can be rewritten using a series of auxiliary relations.

$$\begin{aligned} R_2 &= R; R \\ R_3 &= R_2; R \\ &\cdots \\ R_n &= R_{n-1}; R \\ R^+ &= R \cup R_2 \cup \ldots \cup R_n \end{aligned}$$

These statements can be axiomatized to MSFOL as

$$\begin{aligned} &\forall x, y \bullet R_2(x, y) \Leftrightarrow \exists c \bullet R(x, c) \wedge R(c, y) \\ &\forall x, y \bullet R_3(x, y) \Leftrightarrow \exists c \bullet R_2(x, c) \wedge R(c, y) \cdots \\ &\forall x, y \bullet R_n(x, y) \Leftrightarrow \exists c \bullet R_{n-1}(x, c) \wedge R(c, y) \\ &\forall x, y \bullet R^+(x, y) \Leftrightarrow R(x, y) \vee \bigvee_{i=2}^{n} R_i(x, y) \end{aligned}$$

The reflexive transitive closure is then defined in terms of R^+as

$$\forall x, y \bullet R^*(x, y) \Leftrightarrow R^+(x, y) \vee x = y \tag{4.6}$$

This method requires $n-1$ auxiliary functions and n formulas, and it limits the nesting of quantifiers in the additional axioms to depth three.

4.2.2 Iterative Squaring

Burch et al. [7] describe a method called iterative squaring to define R^+using only $\lceil \log n \rceil$ formulas. Instead of using $n-1$ auxiliary relations to represent paths of a fixed length, Burch et al. use $\lceil \log n \rceil$ auxiliary relations to represent all paths up to and including a given length. Given auxiliary relation R_i that represents all paths in R of length $x \leq 2^i$, the auxiliary relation R_{i+1} that represents all paths in R of length $x \leq 2^{i+1}$ is defined as

$$R_{i+1} = R_i \cup R_i; R_i$$

which can be axiomatized as

$$\forall x, y \bullet R_{i+1}(x, y) \Leftrightarrow R_i(x, y) \vee \exists z \bullet R_i(x, z) \wedge R_i(z, y)$$

The auxiliary relation R_i represents all paths of length $\leq 2^i$ in R. We define auxiliary relations for $i = 1$ through $i = \lceil \log n \rceil$, where $R_0 = R$ and $R_{\lceil \log n \rceil} = R^+$.

The closure operators are encoded as

$$\forall x, y \bullet R^+(x, y) \Leftrightarrow R_{\lceil \log n \rceil}(x, y) \vee \exists z \bullet R_{\lceil \log n \rceil}(x, z) \wedge R_{\lceil \log n \rceil}(z, y)$$
$$\forall x, y \bullet R^*(x, y) \Leftrightarrow x = y \vee R^+(x, y)$$

This method uses $\lceil \log n \rceil$ auxiliary relations and $\lceil \log n \rceil + 2$ formulas. This method has a maximum quantifier depth of three, the same as the simple iterative method.

4.2.3 Claessen's Method

Claessen was the first to introduce an axiomatization of *reflexive* transitive closure in first-order logic over finite domains, rather than the transitive closure [8]. This method requires a constant number of formulas and auxiliary functions, regardless of scope size. Claessen introduces two auxiliary functions:

- $s : A \times A \to A$ which represents the "next step" along the path from the first input to the second
- $C : A \times A \times A \to Bool$ where $C(a, x, z)$ is true if and only if x is closer to z than a

For example, given $R = \{(a, b), (b, c), (c, d), (x, y)\}$,

$$C = \{(a, b, b), (a, b, c), (a, b, d), (a, c, c), (a, c, d), (b, c, c), (b, c, d), (b, d, d), (c, d, d), (x, y, y)\}$$

The "next step" function s must be a total function, so when a path does not exist between two values x and y, $s(x, y)$ can be any arbitrary value. The pairs for which s must be defined are

$$\{(a, b) \to b, (a, c) \to b, (a, d) \to b, (b, c) \to c, (b, d) \to c, (x, y) \to y\} \subset s$$

Using these auxiliary functions, Claessen axiomatizes the reflexive transitive closure R^* of R using nine formulas. First,

$$\forall x, y, z, u : A \bullet C(x, y, u) \wedge C(y, z, u) \;\Rightarrow\; C(x, z, u) \tag{4.7}$$
$$\forall x, y : A \bullet \neg C(x, x, y) \tag{4.8}$$

where Equation 4.7 means that "closeness" to a given point u is transitive, and Equation 4.8 means that any point x is never closer than itself to a point y.

$$\forall x, y : A \bullet R^*(x, y) \wedge \neg(x = y) \;\Rightarrow\; R(x, s(x, y)) \tag{4.9}$$

Equation 4.9 states that if two distinct points x and y are in R^*, then x and the next step on the path from x to y ($s(x, y)$) must be in R.

$$\forall x, y : A \bullet R^*(x, y) \wedge \neg(x = y) \;\Rightarrow\; C(x, s(x, y), y) \tag{4.10}$$
$$\forall x, y : A \bullet R^*(x, y) \wedge \neg(x = y) \;\Rightarrow\; R^*(s(x, y), y) \tag{4.11}$$

Equation 4.10 and Equation 4.11 state that if a non-reflexive pair (x, y) is in R^*, the next step from x towards y is closer to y than x and there is a path from that next point to y respectively.

$$\forall x, y, z : A \bullet R^*(x, y) \wedge R^*(y, z) \;\Rightarrow\; R^*(x, z) \tag{4.12}$$
$$\forall x, y : A \bullet R(x, y) \;\Rightarrow\; R^*(x, y) \tag{4.13}$$
$$\forall x : A \bullet R^*(x, x) \tag{4.14}$$

Lastly, the reflexive transitive closure is defined: Equation 4.12 axiomatizes Equation 4.2, Equation 4.13 axiomatizes Equation 4.1, and Equation 4.14 ensures that R^* is reflexive.

The transitive closure is defined in terms of the reflexive transitive closure.

$$\forall x, y : A \bullet R^+(x, y) \Leftrightarrow \exists z : A \bullet R(x, z) \wedge R^*(z, y) \tag{4.15}$$

Claessen's method uses two auxiliary functions and nine formulas with a maximum quantifier depth of four.

4.2.4 van Eijck's Method

Inspired by Claessen's work, van Eijck provided an alternative axiomatization of reflexive transitive closure [32]. Van Eijck's axiomatization uses a single auxiliary function $C : A \times A \times A \to \mathit{Bool}$ where $C(a, x, z)$ is true if and only if x is closer to z than a along the *shortest* path between a and z. In this way, the auxiliary functions C and s from Claessen's method are combined into one auxiliary function. The reflexive transitive closure is then axiomatized in fewer formulas:

$$\forall x, y, z, u : A \bullet C(x, y, u) \wedge C(y, z, u) \;\Rightarrow\; C(x, z, u) \tag{4.16}$$

$$\forall x, y : A \bullet \neg C(x, x, y) \tag{4.17}$$

As in Claessen's method, Equation 4.16 and Equation 4.17 axiomatize that being closer to the third point is transitive and a point is never closer than itself to a point. Trivially, the end of a path is closer to itself than the start of the path (when the start and end are distinct), so $C(x, y, y)$ means that a path from x to y exists in R.

$$\forall x, y : A \bullet R(x, y) \wedge \neg(x = y) \;\Rightarrow\; C(x, y, y) \tag{4.18}$$

Equation 4.18 ensures every path of length one in R is included in C by ensuring every non-reflexive edge (x, y) is included in C as $C(x, y, y)$.

$$\forall x, y, z : A \bullet C(x, y, y) \wedge C(y, z, z) \wedge \neg(x = z) \;\Rightarrow\; C(x, z, z) \tag{4.19}$$

Equation 4.19 axiomatizes the fact that if a path exists from x to y and y to z (and $x \neq z$), then a path must exist from x to z.

$$\forall x, y, z : A \bullet C(x, y, z) \wedge \neg(y = z) \;\Rightarrow\; C(y, z, z) \tag{4.20}$$

Equation 4.20 describes the fact that for every point y between x and z along the shortest path from x to z, there must be a path from y to z.

$$\forall x, y : A \bullet C(x, y, y) \;\Rightarrow\; \exists z : A \bullet R(x, z) \wedge C(x, z, y) \tag{4.21}$$

Equation 4.21 axiomatizes the fact that, given a path from x to y ($C(x, y, y)$), there must be some edge in R from x to a point z that is closer to y than x is.

$$\forall x, y : A \bullet R^*(x, y) \Leftrightarrow C(x, y, y) \vee x = y \tag{4.22}$$

Lastly, Equation 4.22 defines the reflexive transitive closure as exactly all pairs (x, y) such that a path exists from x to y ($C(x, y, y)$) or $x = y$ because R^* is reflexive.

The transitive closure is defined, as in Claessen's axiomatization, using Equation 4.15.

Van Eijck's method requires only a single auxiliary function and only eight formulas, with a maximum quantifier depth of four.

4.2.5 Liu et al.'s Method

Liu et al. proposed an axiomatization independent of scope size for reflexive transitive closure using integers [22] based on Claessen's work. Liu et al. introduce auxiliary function $P : A \times A \to Int$ where $P(x, y)$ is the length of the shortest path from x to y. Note that P also includes paths that start and end at the same point, and thus are technically not paths, but we will refer to them as paths for the remainder of this chapter. For example, if $(a, a) \in R$ then $P(a, a) = 1$.

P is axiomatized in three formulas

$$\forall x, y : A \bullet R(x, y) \Leftrightarrow P(x, y) = 1 \tag{4.23}$$

$$\forall x, y, z : A \bullet P(x, y) > 0 \land P(y, z) > 0 \;\Rightarrow\; P(x, z) > 0 \tag{4.24}$$

$$\forall x, y : A \bullet P(x, y) > 1 \;\Rightarrow\; \exists z : A \bullet P(x, z) = 1 \land P(x, y) = P(z, y) + 1 \tag{4.25}$$

Where Equation 4.23 states that if an edge between two points exists in R, then the length of the shortest path between those points is one. If $P(x, y) > 0$, there exists a path from x to y in R. So, Equation 4.24 axiomatizes the fact that if a path exists from x to y and y to z then a path exists from x to z. Equation 4.25 states that if the shortest path between x and y is longer than a single edge, then there must be a point z one edge from x along the shortest path from x to y.

The transitive closure is encoded as every value where P is positive.

$$\forall x, y : A \bullet R^+(x, y) \Leftrightarrow P(x, y) > 0 \tag{4.26}$$

The reflexive transitive closure also includes every case where the start and endpoint are the same.

$$\forall x, y : A \bullet R^*(x, y) \Leftrightarrow R^+(x, y) \lor x = y \tag{4.27}$$

Liu et al.'s method uses one auxiliary function and only five formulas, with a maximum quantifier depth of three. However, this method relies on some encoding of integers

4.3 Negative Transitive Closure

In the context of model checking, Vakili and Day [31] identified a subset of computational tree logic (CTL) [15], called CTL-Live, which contains temporal operators commonly used

to express liveness properties and is expressible in FOL. The basis for their method is an encoding of transitive closure to check the validity of $\Delta \models R^+(a,b)$.

The first two parts of the definition of transitive closure (Equation 4.1 and Equation 4.2) are not a complete definition of transitive closure, but they are sufficient for certain problems. We use $\text{defTC}(R,T)$ to describe the incomplete axiomatization of R^+ as relation T, where, without loss of generality, T is a fresh symbol, not used elsewhere.

$$\begin{aligned}\text{defTC}\,(R,T) =&(\forall a,b \bullet (a,b) \in R \;\Rightarrow\; (a,b) \in T)\\ &\wedge (\forall a,b \bullet (a,b) \in T;T \;\Rightarrow\; (a,b) \in T)\end{aligned}$$

Written in MSFOL,

$$\begin{aligned}\text{defTC}\,(R,T) =&(\forall a,b \bullet R(a,b) \;\Rightarrow\; T(a,b))\\ &\wedge (\forall a,b,c \bullet T(a,b) \wedge T(b,c) \;\Rightarrow\; T(a,c))\end{aligned}$$

Vakili [30] shows that this incomplete axiomatization defining the transitive closure of a relation is sufficient for checking the validity of whether a pair is in a transitive closure. Because R^+ is a subset of all the possible values for T in Equation 4.3, T can contain pairs not in R^+ but it cannot omit a pair (i.e., $(a,b) \notin T$ and $(a,b) \in R^+$ is impossible). Vakili [30] proved that given a set of MSFOL formulas Δ,

$$\Delta \models R^+(a,b) \quad \Leftrightarrow \quad \Delta \cup \{\text{defTC}(R,T)\} \models T(a,b) \tag{4.28}$$

Checking the validity of $\Delta \models R^+(a,b)$ is the same as checking that the satisfiability of $\Delta \cup \{\neg R^+(a,b)\}$ is UNSAT. For the rest of this section we will examine this encoding of transitive closure through the lens of satisfiability, rather than validity.

Vakili's work focused on the special case of checking satisfiability where R^+ is used negatively in conjunction with a set of other formulas Δ in the form $\Delta \cup \{\neg R^+(a,b)\}$. We prove that this axiomatization of transitive closure is more generally applicable. Let $\text{TC}(\phi, R)$ be the transformation defined as:

$$\text{TC}\,(\phi, R) = \phi\left[R^+ \middle| T\right] \wedge \text{defTC}(R,T)$$

where ϕ is a term in negation normal form and T is a fresh symbol that does not appear in ϕ. Next, we present a proof that $\text{TC}(\phi, R)$ is equisatisfiable to ϕ under the conditions that

1. membership in R only occurs negatively (i.e., $\neg R^+\,(a,b)$) in the formula

2. ϕ is in negation normal form (NNF)

3. R^+ does not appear in the conditional of an ITE term or an argument of an uninterpreted function

4. R^+ does not appear in the body of an interpreted function definition

We denote a formula ϕ that satisfies these conditions for a relation R as $applicable(\phi, R)$. The formula ϕ must be in NNF because $\neg((a,b) \notin R^+)$ is equivalent to $(a,b) \in R^+$ and our transformation TC is only applicable to negative membership in R. Similarly, if R^+ appears in the conditional of an ITE term or the argument of an uninterpreted function, a T including a pair that is not in R could affect the satisfiability of top-level formula. For example,

$$\text{ITE}(\neg R^+(a,b), false, true)$$

and

$$\forall x : Bool \bullet f(x) = \neg x \wedge f(\neg R^+(a,b))$$

are equivalent to $(a,b) \in R$.

Theorem 4.1. *If* $applicable(\phi, R)$*, then* $\phi \overset{\text{SAT}}{=} \text{TC}(\phi, R)$.

We break apart the proof of Theorem 4.1 into two parts: when $applicable(\phi, R)$, ϕ is SAT implies $\text{TC}(\phi, R)$ is SAT (Theorem 4.2) and vice versa (Theorem 4.3).

Let ϕ be an arbitrary MSFOL formula and R be a relation.

Theorem 4.2. *If* $applicable(\phi, R)$ *and* ϕ *is SAT, then* $\text{TC}(\phi, R)$ *is SAT.*

Proof of Theorem 4.2. We assume $applicable(\phi, R)$ and ϕ is SAT. Therefore, there exists some $\mathcal{I}$ such that $\mathcal{I} \Vdash \phi$. We begin by expanding $\text{TC}(\phi, R)$ to $\phi\,[R^+|T] \wedge \text{defTC}(R, T)$ where T is a fresh symbol. We define $\mathcal{I}'$ to be the interpretation $\mathcal{I}$ extended with the value assignment $(T \to (R^{\mathcal{I}})^+)$. So, $\mathcal{I}' = \mathcal{I}\left[T \middle| (R^{\mathcal{I}})^+\right]$. Because $T^{\mathcal{I}'} = (R^{\mathcal{I}})^+$ and $\text{defTC}(R, T)$ is a subset of the axioms that assert $T = (R^{\mathcal{I}})^+$, we know $\mathcal{I}' \Vdash \text{defTC}(R, T)$.

We prove if $applicable(\phi, R)$ and $\mathcal{I} \Vdash \phi$, then $\mathcal{I}' \Vdash \phi\,[R^+|T] \wedge \text{defTC}(R, T)$ by structural induction on ϕ.

Case 1 (Base Case 1: R^+ is not in ϕ)**.**
T is not present in ϕ, so

$$\phi^{\mathcal{I}} = \phi^{\mathcal{I}'}$$

As $\mathcal{I} \Vdash \phi$, this means $\mathcal{I}' \Vdash \phi$. As R^+ is not in ϕ,

$$\phi = \phi \left[R^+ \middle| T\right] \tag{4.29}$$

Therefore, we can use Equation 4.29 to substitute for ϕ in $\mathcal{I}' \Vdash \phi$, giving us

$$\mathcal{I}' \Vdash \phi \left[R^+ \middle| T\right]$$

As we know $\mathcal{I}' \Vdash \text{defTC}(R, T)$,

$$\mathcal{I}' \Vdash \phi \left[R^+ \middle| T\right] \wedge \text{defTC}(R, T)$$

Case 2 (Base Case 2: $\phi = \neg R^+(A, B)$)**.**
Recall that $\mathcal{I}' \Vdash \text{defTC}(R, T)$.

We apply the substitution in $\phi\,[R^+|T]$ to get

$$\left(\neg R^+(A, B)\right) \left[R^+ \middle| T\right] = \neg T(A \left[R^+ \middle| T\right], B \left[R^+ \middle| T\right]) \tag{4.30}$$

From $applicable(\phi, R)$, we know that R^+ is not in A or B. Therefore, $A = A\,[R^+|T]$ and $B = B\,[R^+|T]$. We substitute into Equation 4.30 to get

$$\left(\neg R^+(A, B)\right) \left[R^+ \middle| T\right] = \neg T(A, B)$$

Because $\mathcal{I} \Vdash \neg R^+(A, B)$,

$$\left(\neg R^+(A, B)\right)^{\mathcal{I}} = \neg \left(R^{\mathcal{I}}\right)^+ (A^{\mathcal{I}}, B^{\mathcal{I}}) = true \tag{4.31}$$

As R^+ is not in A or B, we know that $A^{\mathcal{I}} = A^{\mathcal{I}'}$ and $B^{\mathcal{I}} = B^{\mathcal{I}'}$. Substituting into Equation 4.31 yields

$$\neg \left(R^{\mathcal{I}}\right)^+ (A^{\mathcal{I}'}, B^{\mathcal{I}'}) = true$$

Because $\mathcal{I}'$ maps T to $\left(R^{\mathcal{I}}\right)^+$, $T^{\mathcal{I}'} = \left(R^{\mathcal{I}}\right)^+$. So,

$$\neg T^{\mathcal{I}'}(A^{\mathcal{I}'}, B^{\mathcal{I}'}) = true \tag{4.32}$$

Equation 4.32 means that $\mathcal{I}' \Vdash \neg T(A, B)$. Thus, $\mathcal{I}' \Vdash \neg T(A, B) \wedge \text{defTC}(R, T)$ and

$$\mathcal{I}' \Vdash \left(\neg R^+(A, B)\right) \left[R^+ \middle| T\right] \wedge \text{defTC}(R, T)$$

Case 3 ($\phi = A \wedge B$)**.**
$\mathcal{I} \Vdash A \wedge B$ implies $\mathcal{I} \Vdash A$ and $\mathcal{I} \Vdash B$. As $\mathit{applicable}(A \wedge B)$, both $\mathit{applicable}(A)$ and $\mathit{applicable}(B)$ must hold. By the inductive hypothesis, there exist $\mathcal{I}_A$ and $\mathcal{I}_B$ such that

$$\mathcal{I}_A \Vdash A\left[R^+ \middle| T_A\right] \wedge \mathrm{defTC}(R, T_A) \text{ and } \mathcal{I}_B \Vdash B\left[R^+ \middle| T_B\right] \wedge \mathrm{defTC}(R, T_B) \tag{4.33}$$

where T_A and T_B are fresh, unique symbols. Therefore,

$$\left(A\left[R^+ \middle| T_A\right]\right)^{\mathcal{I}_A} \wedge \left(\mathrm{defTC}(R, T_A)\right)^{\mathcal{I}_A} = \mathit{true} \tag{4.34}$$
$$\left(B\left[R^+ \middle| T_B\right]\right)^{\mathcal{I}_B} \wedge \left(\mathrm{defTC}(R, T_B)\right)^{\mathcal{I}_B} = \mathit{true} \tag{4.35}$$

We construct $\mathcal{I}_{AB}$ to be the interpretation $\mathcal{I}$ extended by the value assignments

$$\left\{\left(T_A \to \left(R^{\mathcal{I}}\right)^+\right), \left(T_B \to \left(R^{\mathcal{I}}\right)^+\right)\right\}$$

By the construction of $\mathcal{I}_{AB}$ we know that

$$T_A^{\mathcal{I}_{AB}} = T_B^{\mathcal{I}_{AB}} \tag{4.36}$$

From Equation 4.36 we can conclude that

$$\left(\mathrm{defTC}(R, T_A)\right)^{\mathcal{I}_{AB}} = \left(\mathrm{defTC}(R, T_B)\right)^{\mathcal{I}_{AB}} \tag{4.37}$$

and

$$\left(B\left[R^+ \middle| T_B\right]\right)^{\mathcal{I}_{AB}} = \left(B\left[R^+ \middle| T_A\right]\right)^{\mathcal{I}_{AB}} \tag{4.38}$$

which we will use later in this proof.

As T_B is not in $A\,[R^+|T_A] \wedge \mathrm{defTC}(R, T_A)$ and T_A is not in $B\,[R^+|T_B] \wedge \mathrm{defTC}(R, T_B)$

$$\left(A\left[R^+ \middle| T_A\right]\right)^{\mathcal{I}_{AB}} \wedge \left(\mathrm{defTC}(R, T_A)\right)^{\mathcal{I}_{AB}} = \mathit{true} \tag{4.39}$$
$$\left(B\left[R^+ \middle| T_B\right]\right)^{\mathcal{I}_{AB}} \wedge \left(\mathrm{defTC}(R, T_B)\right)^{\mathcal{I}_{AB}} = \mathit{true} \tag{4.40}$$

Therefore,

$$\begin{aligned}&\left(A\left[R^+ \middle| T_A\right]\right)^{\mathcal{I}_{AB}} \wedge \left(\mathrm{defTC}(R, T_A)\right)^{\mathcal{I}_{AB}} \\ \wedge\ &\left(B\left[R^+ \middle| T_B\right]\right)^{\mathcal{I}_{AB}} \wedge \left(\mathrm{defTC}(R, T_B)\right)^{\mathcal{I}_{AB}} = \mathit{true}\end{aligned} \tag{4.41}$$

Because of Equation 4.37, Equation 4.41 becomes

$$\left(A\left[R^+ \middle| T_A\right]\right)^{\mathcal{I}_{AB}} \wedge \left(B\left[R^+ \middle| T_B\right]\right)^{\mathcal{I}_{AB}} \wedge \left(\mathrm{defTC}(R, T_A)\right)^{\mathcal{I}_{AB}} = \mathit{true} \tag{4.42}$$

Using Equation 4.38 we can make the following simplifications

$$(A\left[R^+\middle|T_A\right])^{\mathcal{I}_{AB}} \wedge (B\left[R^+\middle|T_A\right])^{\mathcal{I}_{AB}} \wedge (\text{defTC}(R,T_A))^{\mathcal{I}_{AB}} = true \tag{4.43}$$

$$\left((A \wedge B)\left[R^+\middle|T_A\right]\right)^{\mathcal{I}_{AB}} \wedge (\text{defTC}(R,T_A))^{\mathcal{I}_{AB}} = true \tag{4.44}$$

Now, as T_B is not in Equation 4.44, we can drop it from our interpretation and conclude that

$$\left((A \wedge B)\left[R^+\middle|T_A\right]\right)^{\mathcal{I}_A} \wedge ((\text{defTC}(R,T_A)))^{\mathcal{I}_A} = true$$

$\mathcal{I}_A$ is $\mathcal{I}'$ when T_A is T, so

$$\mathcal{I}' \Vdash (A \wedge B)\left[R^+\middle|T_A\right] \wedge (\text{defTC}(R,T_A))$$

Case 4 ($\phi = A \vee B$)**.**
From *applicable*$(A \vee B)$, we know *applicable*(A) and *applicable*(B). We know $\mathcal{I} \Vdash A \vee B$, which means $\mathcal{I} \Vdash A$ or $\mathcal{I} \Vdash B$. Without loss of generality, we choose $\mathcal{I} \Vdash A$. We know that for the interpretation $\mathcal{I}'$ (which extends $\mathcal{I}$ with the assignment $T \to R^+$) $\mathcal{I}' \Vdash A\,[R^+|T] \wedge \text{defTC}(R,T)$ by the inductive hypothesis. Therefore, $\mathcal{I}' \Vdash A\,[R^+|T]$ and $\mathcal{I}' \Vdash \text{defTC}(R,T)$ separately. Thus,

$$\mathcal{I}' \Vdash A\left[R^+\middle|T\right] \vee B\left[R^+\middle|T\right] \tag{4.45}$$

We simplify Equation 4.45 to

$$\mathcal{I}' \Vdash (A \vee B)\left[R^+\middle|T\right]$$

Thus,

$$\mathcal{I}' \Vdash (A \vee B)\left[R^+\middle|T\right] \wedge \text{defTC}(R,T) \tag{4.46}$$

□

A case for $\phi = \neg A$ is covered by the above cases because ϕ is in negation normal form and *applicable*(ϕ, R) means Case 1 and Case 2 are the only places a negation can appear in ϕ. We do not need cases for the universal and existential quantifiers because these are equivalent to a finite conjunction and disjunction respectively.

We now prove satisfiability in the opposite direction.

Theorem 4.3. *If* *applicable*(ϕ, R) *and* TC(ϕ, R) *is SAT, then* ϕ *is SAT.*

If $\text{TC}(\phi, R)$ is SAT, there exists some interpretation $\mathcal{I}_v$ that maps T to a value v and $\mathcal{I}_v \Vdash \text{TC}(\phi, R)$. We cannot assume v is R^+ for this direction of the proof. Let $\mathcal{I}$ be the interpretation $\mathcal{I}_v$ without a value assignment assignment for T. Let $\mathcal{I}'$ be an interpretation that extends $\mathcal{I}$ with the assignment $T \to \left(R^{\mathcal{I}}\right)^+$.

We utilize two lemmas, which we will prove below, to aid in the proof of Theorem 4.3.

Lemma 4.3.1. *If* $\textit{applicable}(\phi, R)$ *and* $\mathcal{I}_v \Vdash \phi\left[R^+|T\right] \wedge \text{defTC}(R, T)$*, then*

$$\mathcal{I}' \Vdash \phi\left[R^+\middle|T\right] \wedge \text{defTC}(R, T)$$

Lemma 4.3.2. *If* $\textit{applicable}(\phi, R)$ *and* $\mathcal{I}' \Vdash \phi\left[R^+|T\right] \wedge \text{defTC}(R, T)$*, then* $\mathcal{I} \Vdash \phi$*.*

Proof of Theorem 4.3. We are given $\textit{applicable}(\phi, R)$ and $\text{TC}(\phi, R)$ is SAT. Because $\text{TC}(\phi, R)$ is SAT, we know there exists some $\mathcal{I}_v$ that maps T to a value v and $\mathcal{I}_v \Vdash \text{TC}(\phi, R)$. We expand TC to obtain

$$\mathcal{I}_v \Vdash \phi\left[R^+\middle|T\right] \wedge \text{defTC}(R, T)$$

By Lemma 4.3.1, we then know

$$\mathcal{I}' \Vdash \phi\left[R^+\middle|T\right] \wedge \text{defTC}(R, T)$$

By Lemma 4.3.2, we can conclude $\mathcal{I} \Vdash \phi$. As there exists an interpretation $\mathcal{I}$ that satisfies ϕ, ϕ is SAT. □

We now prove the lemmas utilized in our proof of Theorem 4.3. Recall

Lemma 4.3.1. *If* $\textit{applicable}(\phi, R)$ *and* $\mathcal{I}_v \Vdash \phi\left[R^+|T\right] \wedge \text{defTC}(R, T)$*, then*

$$\mathcal{I}' \Vdash \phi\left[R^+\middle|T\right] \wedge \text{defTC}(R, T)$$

Proof of Lemma 4.3.1. Since $\mathcal{I}_v \Vdash \phi\left[R^+|T\right] \wedge \text{defTC}(R, T)$, we know that $\mathcal{I}_v \Vdash \text{defTC}(R, T)$. Because $\mathcal{I}_v \Vdash \text{defTC}(R, T)$, T must be transitive and contain every pair in R. As R^+ is the smallest relation that satisfies those two conditions, we know that $\left(R^{\mathcal{I}}\right)^+ \subseteq T^{\mathcal{I}_v}$. Therefore, if any pair is not in $T^{\mathcal{I}_v}$, that pair is not in $\left(R^{\mathcal{I}}\right)^+$. Because in $\mathcal{I}'$, T is mapped to $\left(R^{\mathcal{I}}\right)^+$, $\mathcal{I}' \Vdash \text{defTC}(R, T)$. Additionally, as $\mathcal{I}'$ explicitly maps T to $\left(R^{\mathcal{I}}\right)^+$,

$$T^{\mathcal{I}'} = \left(R^{\mathcal{I}}\right)^+ \tag{4.47}$$

We prove Lemma 4.3.1 by structural induction on ϕ.

Case 1 (Base Case 1: ϕ does not contain R^+)**.**

$$\mathcal{I}_v \Vdash \phi\left[R^+\middle|T\right] \wedge \text{defTC}(R,T)$$

So,

$$\mathcal{I}_v \Vdash \phi\left[R^+\middle|T\right]$$

Given ϕ does not contain R^+, we know that $\phi = \phi\left[R^+|T\right]$. Therefore,

$$\mathcal{I}_v \Vdash \phi$$

$$\phi^{\mathcal{I}_v} = true$$

As ϕ does not contain T, we know that

$$\phi^{\mathcal{I}_v} = \phi^{\mathcal{I}'}$$

Thus,

$$\phi^{\mathcal{I}'} = true$$

and

$$\mathcal{I}' \Vdash \phi$$

We know that $\phi = \phi\left[R^+|T\right]$, so

$$\mathcal{I}' \Vdash \phi\left[R^+\middle|T\right]$$

Allowing us to conclude

$$\mathcal{I}' \Vdash \phi\left[R^+\middle|T\right] \wedge \text{defTC}(R,T)$$

Case 2 (Base Case 2: $\phi = \neg R^+(A,B)$)**.**
We know $\mathcal{I}_v \Vdash \left(\neg R^+(A,B)\right)\left[R^+|T\right]$ and $I_v \Vdash \text{defTC}(R,T)$.

We simplify $\mathcal{I}_v \Vdash \left(\neg R^+(A,B)\right)\left[R^+|T\right]$ to $\mathcal{I}_v \Vdash \neg T(A\left[R^+|T\right], B\left[R^+|T\right])$. As we know $applicable(\phi, R)$, R^+ is not in A or B, so $A\left[R^+|T\right] = A$ and $B\left[R^+|T\right] = B$ allowing us to further simplify $\mathcal{I}_v \Vdash \neg T(A\left[R^+|T\right], B\left[R^+|T\right])$ to

$$\mathcal{I}_v \Vdash \neg T(A,B)$$

Therefore,

$$\neg T^{\mathcal{I}_v}(A^{\mathcal{I}_v}, B^{\mathcal{I}_v}) = true$$

Because we know that if a pair is not in $T^{\mathcal{I}_v}$ then it is not in $\left(R^{\mathcal{I}}\right)^+$, we can conclude

$$\neg\left(R^{\mathcal{I}}\right)^+(A^{\mathcal{I}_v}, B^{\mathcal{I}_v}) = true$$

As A and B both do not contain T, $A^{\mathcal{I}_v} = A^{\mathcal{I}'}$ and $B^{\mathcal{I}_v} = B^{\mathcal{I}'}$. Therefore,

$$\neg\left(R^{\mathcal{I}}\right)^{+}(A^{\mathcal{I}'}, B^{\mathcal{I}'}) = true$$

From Equation 4.47, $\left(R^{\mathcal{I}}\right)^{+} = T^{\mathcal{I}'}$. So,

$$\neg T^{\mathcal{I}'}(A^{\mathcal{I}'}, B^{\mathcal{I}'}) = true$$

Therefore,

$$\mathcal{I}' \Vdash \neg T(A, B)$$

As $\mathcal{I}' \Vdash \text{defTC}(R, T)$,

$$\mathcal{I}' \Vdash \neg T(A, B) \wedge \text{defTC}(R, T)$$

Both A and B do not contain T, so $\neg R^{+}(A, B)\,[R^{+}|T] = \neg T(A, B)$, giving us

$$\mathcal{I}' \Vdash \neg R^{+}(A, B)\left[R^{+}\middle|T\right] \wedge \text{defTC}(R, T)$$

Case 3 ($\phi = A \wedge B$)**.**

$$\begin{aligned}\mathcal{I}_v \Vdash &(A \wedge B)\left[R^{+}\middle|T\right] \wedge \text{defTC}(R, T)\\ \mathcal{I}_v \Vdash &A\left[R^{+}\middle|T\right] \wedge B\left[R^{+}\middle|T\right] \wedge \text{defTC}(R, T)\end{aligned}$$

So, separately,

$$\begin{aligned}\mathcal{I}_v \Vdash &A\left[R^{+}\middle|T\right] \wedge \text{defTC}(R, T)\\ \mathcal{I}_v \Vdash &B\left[R^{+}\middle|T\right] \wedge \text{defTC}(R, T)\end{aligned}$$

applicable(A, R) and *applicable*(B, R) must both hold, as if either does not then *applicable*$(A \wedge B, R)$ must be false. Therefore,

$$\begin{aligned}&\textit{applicable}(A, R) \text{ and } \mathcal{I}_v \Vdash A\left[R^{+}\middle|T\right] \wedge \text{defTC}(R, T)\\ &\textit{applicable}(B, R) \text{ and } \mathcal{I}_v \Vdash B\left[R^{+}\middle|T\right] \wedge \text{defTC}(R, T)\end{aligned}$$

By the inductive hypothesis, this becomes

$$\begin{aligned}\mathcal{I}' \Vdash &A\left[R^{+}\middle|T\right] \wedge \text{defTC}(R, T)\\ \mathcal{I}' \Vdash &B\left[R^{+}\middle|T\right] \wedge \text{defTC}(R, T)\end{aligned}$$

We can combine these statements to prove

$$\mathcal{I}' \Vdash A\left[R^+\middle|T\right] \wedge \text{defTC}(R,T) \wedge B\left[R^+\middle|T\right] \wedge \text{defTC}(R,T)$$

which we can simplify to

$$\begin{aligned}\mathcal{I}' \Vdash & A\left[R^+\middle|T\right] \wedge B\left[R^+\middle|T\right] \wedge \text{defTC}(R,T)\\ \mathcal{I}' \Vdash & (A \wedge B)\left[R^+\middle|T\right] \wedge \text{defTC}(R,T)\end{aligned}$$

Case 4 ($\phi = A \vee B$)**.**

$$\mathcal{I}_v \Vdash (A \vee B)\left[R^+\middle|T\right] \wedge \text{defTC}(R,T)$$

Separately, we obtain

$$\mathcal{I}_v \Vdash \text{defTC}(R,T) \tag{4.48}$$
$$\mathcal{I}_v \Vdash (A \vee B)\left[R^+\middle|T\right] \tag{4.49}$$

We can expand the substitution in Equation 4.49 to obtain

$$\mathcal{I}_v \Vdash A\left[R^+\middle|T\right] \vee B\left[R^+\middle|T\right]$$

which means $\mathcal{I}_v \Vdash A\,[R^+|T]$ or $\mathcal{I}_v \Vdash B\,[R^+|T]$. Without loss of generality, we choose $\mathcal{I}_v \Vdash A\,[R^+|T]$. Given Equation 4.48, we then know

$$\mathcal{I}_v \Vdash A\left[R^+\middle|T\right] \wedge \text{defTC}(R,T)$$

As $\mathit{applicable}(A \vee B, R)$, $\mathit{applicable}(A, R)$ must also be true. If $\mathit{applicable}(A, R)$ was not true, $\mathit{applicable}(A \vee B, R)$ could not be true. By the inductive hypothesis, we know

$$\mathcal{I}' \Vdash A\left[R^+\middle|T\right] \wedge \text{defTC}(R,T)$$

which separates to become

$$\mathcal{I}' \Vdash \text{defTC}(R,T) \tag{4.50}$$
$$\mathcal{I}' \Vdash A\left[R^+\middle|T\right] \tag{4.51}$$

If $\mathcal{I}'$ satsifies a formula, it will satisfy the disjunction of that formula and any other formula. Thus, we can conclude from Equation 4.51 that

$$\mathcal{I}' \Vdash A\left[R^+\middle|T\right] \vee B\left[R^+\middle|T\right]$$

which simplifies to

$$\mathcal{I}' \Vdash (A \vee B) \left[R^+ \middle| T\right] \tag{4.52}$$

Given both Equation 4.52 and Equation 4.50, we conclude

$$\mathcal{I}' \Vdash (A \vee B) \left[R^+ \middle| T\right] \wedge \text{defTC}(R, T)$$

□

Next, we prove

Lemma 4.3.2. *If applicable*(ϕ, R) *and* $\mathcal{I}' \Vdash \phi\,[R^+|T] \wedge \text{defTC}(R, T)$*, then* $\mathcal{I} \Vdash \phi$*.*

Proof of Lemma 4.3.2. We assume $\mathcal{I}' \Vdash \phi\,[R^+|T] \wedge \text{defTC}(R, T)$. Therefore, $\mathcal{I}' \Vdash \phi\,[R^+|T]$. This gives us

$$\left(\phi \left[R^+ \middle| T\right]\right)^{\mathcal{I}'} = \mathit{true} \tag{4.53}$$

We constructed $\mathcal{I}'$ to include the assignment $T \to \left(R^{\mathcal{I}'}\right)^+$ and interpretation $\mathcal{I}$ to be $\mathcal{I}'$ without that assignment. Therefore,

$$T^{\mathcal{I}'} = \left(R^{\mathcal{I}}\right)^+ \text{ and } R^{\mathcal{I}'} = R^{\mathcal{I}}$$

Therefore, $\left(R^{\mathcal{I}}\right)^+ = (R^+)^{\mathcal{I}} = (R^+)^{\mathcal{I}'} = T^{\mathcal{I}'}$. So,

$$\phi^{\mathcal{I}'} = \left(\phi \left[R^+ \middle| T\right]\right)^{\mathcal{I}'}$$

as anywhere T appears in $\phi\,[R^+|T]$, R^+ would appear in ϕ and under $\mathcal{I}'$, T and R^+ have the same value. We use the equality to substitute ϕ for $\phi\,[R^+|T]$ in Equation 4.53 to obtain

$$\phi^{\mathcal{I}'} = \mathit{true}$$

As ϕ does not contain T, $\phi^{\mathcal{I}'} = \phi^{\mathcal{I}}$. Therefore,

$$\phi^{\mathcal{I}} = \mathit{true}$$

$$\mathcal{I} \Vdash \phi$$

□

As we have proved both Theorem 4.2 and Theorem 4.3, we can conclude that Theorem 4.1 is true.

When the transitive closure of a relation is only used negatively, our encoding of transitive closure uses no auxiliary relations and two formulas with a maximum quantifier depth of three.

4.3.1 Use of Negative Transitive Closure Transformation for MSFMF Problems

As the negative transitive closure over any arbitrary relation can be removed and replaced by an equisatisfiable formula via TC, the process can be repeated for every applicable relation to create a formula equisatisfiable to the original without the transitive closure operator. Relations that contain a positive closure ($(a, b) \in R$) can be transformed by other methods.

Our transformation of transitive closure, uses fewer terms than the other, more general, substitutions examined earlier in this chapter. While our method reduces the size of the input to the SMT solver, it is less restrictive on the search space, as it allows the function representing R^+ to be any superset of R^+ instead of exactly R^+, so it may not benefit solving time.

Algorithm 14 Negative Transitive Closure Elimination

Require: ϕ is in NNF
Ensure: $\phi \overset{\text{SAT}}{=} \psi$
  $rels \leftarrow \{r |\ r \text{ is a relation used in } \phi\}$
  $\psi \leftarrow \phi$
  for $r \in rels$ **do**
    if $applicable(\psi, r)$ and r^+ is used in ψ **then**
      $\psi \leftarrow \text{TC}(\psi, r)$
    end if
  end for

4.4 Closure over Relations of Arbitrary Arity

When modeling a system, relations may be dependent on more than just an "input" and an "output". For example, given a relation $Go : Location \times Location \times TransitType \rightarrow Bool$, $Go(a, b, t)$ might represent that there is direct access from location a to location b via a given type of transportation t. $Go(a, b, CAR)$ being false would represent that a car cannot go directly from a to b, but $Go(a, b, BIKE)$ would be true if there is a bike trail or promenade connecting the locations.

To specify that there is a car-based path between two locations a and b, one could

define an auxiliary relation $Go_{CAR} : Location \times Location \rightarrow Bool$ as

$$\forall a, b : Location \bullet Go_{CAR}(a, b) \Leftrightarrow Go(a, b, CAR) \tag{4.54}$$

and represent the existence of a driving path from a to b as simply ${Go_{CAR}}^{+}(a, b)$. However, this method is not always practical, as representing the idea that there is some type of transportation you can use to get from a to b using auxiliary functions like Equation 4.54 would require defining an auxiliary function for every possible $TransitType$. Furthermore, if the relation is of higher arity, an auxiliary function would be needed for each value in the cross product of the fixed arguments, which results in an explosion in the number of formulas needed to axiomatize the fact.

Instead, we extend the axiomatizations of transitive closure to contain the values which will be fixed during the closure (in this case a $TransitMethod$). Given a relation $R : A \times A \times B_1 \times \ldots \times B_n$, every auxiliary function for transitive closure described as $f : C_1 \times \ldots \times C_n$ becomes $f : C_1 \times \ldots \times C_m \times B_1 \times \ldots \times B_n$. Algorithm 15 describes how the axioms defining transitive closure of a relation are generalized for higher order relations.

Algorithm 15 Generalize Axiom ϕ Defining Transitive Closure of R with arity ≥ 2

Require: $R : A \times A \times B_1 \times \ldots \times B_n : Bool$ has arity greater than 2
  Let $b_1, \ldots, b_n$ be fresh symbols
  $\psi \leftarrow \forall b_1 : B_1 \bullet \ldots \forall b_n : B_n \bullet \phi$
  for each function invocation $f(a_1, \ldots, a_m)$ **do**
    $\psi \leftarrow \psi\,[f(a_1, \ldots, a_m) | f(a_1, \ldots, a_m, b_1, \ldots, b_n)]$
  end for

For example, in van Eijck's method, described in Section 4.2.4, a single auxiliary function $C : A \times A \times A \rightarrow Bool$ is used. Using the example, we alter C to be $C : Location \times Location \times Location \times TransitMethod \rightarrow Bool$, where $C(a, x, z, m)$ indicates that x is closer to z than a along the shortest path between a and z *using method* m. Any formula axiomatizing the transitive closure is wrapped in $\forall m : TransitMethod$. For example, Equation 4.18 becomes

$$\forall m : TransitMethod \bullet \forall x, y : Location \bullet R(x, y, m) \wedge \neg(x = y) \;\Rightarrow\; C(x, y, y, m)$$

4.5 Evaluation

Table 4.1 describes the number of formulas added, auxiliary functions added, and the maximum quantifier depth of formulas added for the various methods of axiomatizing

transitive closure. The table also shows if the method requires the use of integers. The number of formulas and auxiliary functions required for the simple iterative method scales linearly with the scope of the sort being closed over and logarithmically for the iterative squaring method. The other methods improve on this with a constant number of formulas and auxiliary functions. Negative transitive closure requires only two additional formulas and zero additional auxiliary functions and is tied for the lowest maximum quantifier depth at three. We have separate columns for the depth of all quantifiers ($\forall\exists$ Depth) and the depth of only universal quantifiers ($\forall$ Depth) because existential quantifiers can be skolemized, avoiding the increase in term size caused by quantifier expansion.

Table 4.1: Syntactic Analysis of Transitive Closure Methods over Sort of Scope n

Method	Formulas	Aux. Functions	$\forall\exists$ Depth	$\forall$ Depth	Ints
Simple Iterative	n	$n-1$	3	2	No
Iterative Squaring	$\lceil \log n \rceil$	$\lceil \log n \rceil$	3	2	No
Claessen's Method	9	2	4	4	No
van Eijck's Method	8	1	4	4	No
Liu et al.'s Method	5	1	3	3	Yes
Negative T.C.	2	0	3	3	No

We implement negative transitive closure (Section 4.3) and the methods discussed in Section 4.2 as transformers in Fortress to compare their performance on MSFMF problems in an SMT solver. These implementations of transitive closure elimination utilize our generalization for transitive closure over higher-arity relations. The transformers are applied after the problem is converted into negation normal form. Any new axioms created are converted into negation normal form before they are added to the problem, to ensure the problem remains in negation normal form.

Liu et al.'s method requires integers to be used in the problem to count every edge in R^+. Liu et al.'s method will never count higher than the scope of the sort being closed over. We use OPFI (Section 3.2) to handle overflows, so we do not actually use bitvectors. For each problem, we must choose a finite scope for integers. We used the range $[-2^{n-1}, 2^{n-1}-1]$, with $n = 4$ as a default. To avoid overflows, we increment n until the scope of every sort in the problem is within the finite range $[-2^{n-1}, 2^{n-1}-1]$.

Because there are no benchmark sets for MSFOL and TC, we gather problems constructed by other members of the WATForm lab at the University of Waterloo using Dash+ that have instances of negative transitive closure to compare the methods. We created a small extension of SMT-LIB, called SMTTC, that includes the transitive and reflexive

transitive closure operators. Dash+ is a modelling language for hierarchical, concurrent state machines implemented as an extension of Alloy [27]. Using the Dash+ translation to Alloy [16] and the translation of Alloy to SMTTC (work in progress), we create 18 problems from existing Dash+ models written in SMTTC, each with at least one use of the negative transitive closure operator. These problems are all UNSAT.

We run the performance evaluation on an Intel ® Xeon ® CPU E3-1240 v5 @ 3.50GHz x 8 machine running Linux version 4.4.0-210-generic with up to 64GB of user-space memory. We average the running time of each method on each problem at scope 5, 7, and 9 for the scope of Dash Snapshots, which represent states in the transition system. We use a 20 minute timeout. The expected result of every problem is UNSAT. We did not consider the simple iterative method as it performs significantly worse than the other methods and frequently ran out of memory in preliminary testing. Some problems contained negative transitive closure over some relations and positive transitive closure over others. We use van Eijck's method to eliminate the transitive closure of any relations used positively after eliminating closures with negative transitive closure. We choose van Eijck's method as it performed consistently and seemed less prone to explosions in running time during preliminary experimentation. Tariq did a small performance evaluation of these methods (including iterative and excluding the negative transitive closure method) on three models, which supports these preliminary results [28].

We perform statistical analysis on the results, considering only the largest scope for each model where no method timed out, or the next highest scope if at least one method did not timeout. Timeouts were treated as the maximum 20 minutes, which biases the dataset in favor of the methods that timed out. The resulting data set is presented in Appendix A. As the times we record are not normally distributed, and had two independent variables (problem and method), we perform a Friedman rank sum test on the running times of the methods across the various problems. The test ranks each method's performance on each problem in order to normalize the data. The null hypothesis for this test is that all of the methods have the same average rank across problems. In these tests, the p-value is the probability our test results occurred, supposing the null hypothesis is true. A small p-value is evidence that the null hypothesis is likely false. If a p-value is smaller than a significance level α, then there is statistically significant evidence (at α) to conclude that the null hypothesis is false. We choose $\alpha = 0.05$. We obtain a p-value of 0.01997, which means that there is statistically significant evidence that not all of the models perform the same over the problems considered.

We perform a pairwise Wilcoxon test with a Bonferroni adjustment as a post-hoc test to compare the transitive closure elimination methods pairwise. As the results in Table 4.2 show, this test did not provide statistically significant evidence that any two particular

Table 4.2: Wilcoxon Post-Hoc p-values for Transitive Closure Elimination Methods

	Liu et al.	Squaring	Claessen	van Eijck
Squaring	1.00	-	-	-
Claessen	0.52	1.00	-	-
van Eijck	0.91	1.00	1.00	-
Negative	0.38	1.00	1.00	1.00

Table 4.3: Quade Post-Hoc p-values for Transitive Closure Elimination Methods

	Liu et al.	Squaring	Claessen	van Eijck
Squaring	0.739	-	-	-
Claessen	0.029	0.051	-	-
van Eijck	0.140	0.194	0.485	-
Negative	0.011	0.015	0.193	0.522

Highlighted cells are significant at $\alpha = 0.05$

methods performed differently than one another.

These statistical tests are limited by their inability to consider the magnitude of the difference in running time, as they rely on ranking the running times to normalize the data. Our results showed various methods ranking and performing very differently. So, we perform a Quade test [11], which is used for the same conditions, but considers the magnitude of the difference in the running times between methods on a given problem. The Quade test gave us a p-value of 0.00351, which indicates that there is statistically significant evidence that at least one of the methods performs differently than the others.

We perform a Quade all-pairs post-hoc test with an adjustment to the p-value for the false discovery rate as our original Quade test provides evidence that at least one method is different. Table 4.3 shows that there is statistically significant evidence that negative transitive closure and Claessen's method each performed differently than Liu et al.'s method and the iterative squaring method. The Quade test is limited in its consideration only of the magnitude of the differences. The test is weak at detecting differences in methods when their performance metrics have a small magnitude of difference, even if one consistently performs better.

As another way of comparing the methods, Table 4.4 displays the sum of the times recorded across models for each transitive closure elimination method we considered at the scopes considered for our statistical analysis. Negative transitive closure is the fastest method by this metric, but this result puts greater emphasis on models with longer running

times.

Table 4.4: Total Running Time for Transitive Closure Methods across All Models

Negative	Claessen	van Eijck	Squaring	Liu et al.
4017	5052	5694	7499	11100

Time in seconds

Negative transitive closure and Claessen's method both clearly outperform iterative squaring and Liu et al.'s methods by this metric. This metric is biased towards methods that perform well on problems that take take longer to solve, as larger proportional differences in faster problems are valued less than smaller proportional differences in problems that take more time to solve.

4.5.1 Threats to Validity

One threat to *internal validity* is that negative transitive closure is not applicable to every relation in every problem. For problems where this is the case, we use van Eijck's method to encode the relations for which negative transitive closure was not applicable. Another threat to *internal validity* is the limited number of problems considered. A threat to *external validity* is the fact that the problems examined were all created from Dash+ models and (at least in part) by members of the WATForm lab at the University of Waterloo. Biases in Dash+ models, or in how members of the WATForm lab create models may limit the broader applicability of these results. Another potential threat to *external validity* is that we consider only UNSAT problems. The encodings considered may behave differently than observed on problems that are SAT. Unfortunately, at this time, to the best of our knowledge, there is not existing benchmark for problems consisting of FOL and transitive closure.

4.6 Summary

Negative transitive closure uses fewer formulas than the other methods for transitive closure elimination considered in this thesis. It additionally requires no auxiliary functions and is tied with several methods for the smallest maximum quantifier depth. We find statistically significant evidence that negative transitive closure and Claessen's method solve the problems considered faster than iterative squaring and Liu et al.'s methods over the

problems considered. When considering every scope cand problem considered, negative transitive closure solves the problems fastest. Negative transitive closure is the fastest method for 7 of 18 problems considered, and spread over the corpus it performs better than the other methods considered.

Chapter 5

Conclusions

This thesis provides improvements to finite representations of integers for MSFMF problems in SMT solvers. This thesis provides a formulation, called no-overflow bitvectors (NOBV), of Milicevic and Jackson's [25] method for preventing overflow when using bitvectors to represent integers in MSFOL. This thesis then presents a number of extensions to NOBV, generalizing it to support additional abstractions present in MSFOL. This thesis also introduces overflow-preventing finite integers (OPFI). OPFI is a method for finitizing integers that utilizes the theory of integers built into SMT solvers. OPFI is theoretically more correct than NOBV (and unchecked bitvectors). This thesis then provides a performance evaluation of unchecked bitvectors, NOBV, and OPFI. This thesis shows OPFI outperforms NOBV in the SMT solver Z3 and does not perform significantly differently than unchecked bitvectors.

This thesis introduces the negative transitive closure method as an encoding of uses of negative transitive closure in an arbitrary MSFOL formula. Negative transitive closure reduces the number of formulas needed to axiomatize the transitive closure of a relation and uses no auxiliary functions. This thesis also provides a generalization of transitive closure over higher-arity relations that avoids using additional auxiliary functions. This thesis provides a performance evaluation of negative transitive closure and several existing axiomatizations of transitive closure over finite domains. Negative transitive closure solved the problems considered overall faster than the other methods considered and is statistically significantly faster than Liu et al.'s method and the iterative squaring method. However, research is needed to look at additional datasets and more problems.

Both OPFI and negative transitive closure depend the polarity of terms when translating them. When terms are contained within function definitions they have the potential to

be used both positively and negatively. In OPFI, if an integer predicate is within a function definition, we axiomatize the function using $\Leftrightarrow$, which allows us to treat the function as positive in one subterm and negative in a separate subterm. Axiomatizing a function definition containing a closure will not allow negative transitive closure to eliminate that closure. While OPFI needs to know the polarity of a term, it still functions regardless of what that polarity is. Negative transitive closure, conversely, requires that the polarity of terms of the form $(x, y) \in R^+$ are negative. Therefore, axiomatizing the function definition using $\Leftrightarrow$ will guarantee that the relation in consideration is used both positively and negatively, preventing negative transitive closure from being used to axiomatize it.

5.1 Future Work

There are several opportunities for improving MSFMF using SMT solvers.

5.1.1 Portus

Work is currently ongoing to translate Alloy models to Fortress for MSFMF via a tool called Portus built within the Alloy Analyzer. Both integers and the transitive closure operator are commonly used in Alloy models [13]. The improvements this thesis provides in representing integers and transitive closure are useful for efficiently encoding Alloy models in MSFOL for SMT solvers.

5.1.2 OPFI for Natural Numbers

Bitvectors represent values in a fixed range $[-2^{n-1}, 2^{n-1} - 1]$ for a value n. However, negative values may not be needed to express certain models. For example, Liu et al.'s axiomatization of transitive closure and counting the number of values for which a predicate is true both require only non-negative integer values. As a result, half of the search space is unused. OPFI can represent the range $[0, 2^n - 1]$ just as easily as it represents $[-2^{n-1}, 2^{n-1} - 1]$.

5.1.3 Reduction of Overflow Checking in OPFI

Currently, OPFI checks both arguments to the built-in integer predicates ($=, <, >$, etc.) to ensure overflows do not occur in equations like

$$\forall x \bullet \exists y \bullet x + 1 < y$$

because values can only be chosen from inside the range of integers. In the above example, when $x = MAX$, there does not exist a y in $[MIN, MAX]$ such that $x + 1 < y$ is true. Without checking the arguments to the integer predicate for overflows, this would incorrectly evaluate to false. If both arguments do not contain any quantified integers, either in the form of variables or uninterpreted functions, this problem does not occur. However, OPFI will still look for an overflow in cases like

$$MAX + 1 = MAX + 1$$

which incorrectly evaluates to false under OPFI. Detecting these unquantified terms to avoid checking them for overflow could increase the correctness of OPFI.

5.1.4 Hybrid Bounded and Unbounded Integer Problems

In our performance evaluation of bounded integer methods on the SMT-LIB UFNIA benchmark set, unbounded integers solved problems faster than the bounded methods. However, SMT solvers cannot solve some problems for unbounded integers. One could replace some instances of unbounded integers with finitized integers while leaving others unbounded. This would yield more theoretically correct results and potentially faster than converting every usage of unbounded integers to bounded integers.

5.1.5 Solver-Specific Implementations of Transitive Closure

Currently, Fortress can use any off-the-shelf SMT solver that can take input in the SMT-LIB format. Individual solvers perform differently and can support different theories and operations directly. Notably, Z3 supports transitive closure of binary relations [34]. Fortress may benefit from using a direct reference to transitive closure in Z3's API. However, this would require using a different generalization for closure over higher-arity.

5.1.6 SMT Algebraic Datatypes

For our evaluations of integer methods and transitive closure, Fortress uses distinct constants to represent sort elements in SMT. Quantifiers are removed from the problem, to produce a decidable problem in EUF, through skolemization and quantifier expansion. However, SMT solvers have added algebraic datatypes. If algebraic datatypes are used to represent domain elements, quantifiers may not need to be removed from the problem because there is a finite search space for the quantified variables. Methods for transitive closure elimination may perform differently when using algebraic datatypes rather than distinct constants to encode domain elements.

References

[1] Wilhelm Ackermann. *Solvable Cases of the Decision Problem*. North-Holland, 1954.

[2] Clark Barrett, Pascal Fontaine, and Aaron Stump. The SMT-LIB Standard: Version 2.6, July 2017.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[4] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, Switzerland, 2018.

[5] Armin Biere and Daniel Kröning. *SAT-Based Model Checking*, pages 277–303. Springer International Publishing, Cham, Switzerland, 2018.

[6] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'06, pages 427–442, Berlin, Heidelberg, 2006. Springer-Verlag.

[7] Jerry R Burch, Edmund M Clarke, et al. Symbolic model checking: 10^{20} states and beyond. *Information and computation*, 98(2):142–170, 1992.

[8] Koen Claessen. Expressing transitive closure for finite domains in first-order logic. *Unpublished Draft*, 2008.

[9] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style finite model finding. In *Conference on Automated Deduction*, pages 11–27, 2003.

[10] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. *Introduction to Model Checking*, pages 1–26. Springer International Publishing, Cham, Switzerland, 2018.

[11] W. J. Conover. *Practical nonparametric statistics*. Wiley, 1999. Quade.

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

[13] Eid, Elias. Profiling Alloy models. Master's thesis, University of Waterloo, David R. Cheriton School of Computer Science, 2021.

[14] Keith Ellis and Daniel M. Berry. Quantifying the impact of requirements definition and management process maturity on project outcome in large business application development. *Requirements Engineering*, 18(3):223–249, Sep 2013.

[15] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, December 1982.

[16] Tamjid Hossain and Nancy A. Day. Dash+: Extending Alloy with hierarchical states and replicated processes for modelling transition systems. In *International Workshop on Model-Driven Requirements Engineering (MoDRE) @ IEEE International Requirements Engineering Conference (RE)*. IEEE, 2021.

[17] Daniel Jackson. An intermediate design language and its analysis. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '98/FSE-6, page 121–130, New York, NY, USA, 1998. Association for Computing Machinery.

[18] Daniel Jackson. *Software Abstractions*. MIT Press, revised edition, 2016.

[19] R.B. Jones, D.L. Dill, and J.R. Burch. Efficient validity checking for processor verification. In *Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*. IEEE Comput. Soc. Press, November 1995.

[20] Laura Kovács and Andrei Voronkov. First-order theorem proving and VAMPIRE. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[21] Daniel Kroening and Ofer Strichman. *Bit Vectors*, pages 135–156. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[22] Tianhai Liu, Michael Nagel, and Mana Taghdiri. Bounded program verification using an SMT solver: A case study. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, page 101–110, USA, 2012. IEEE Computer Society.

[23] William McCune. A Davis-Putnam program and its application to finite first-order model search: Quasigroup Existence Problem. Technical report, 1994.

[24] William McCune. Mace4 Reference Manual and Guide. Technical Report arXiv: cs/0310055, October 2003.

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

[26] Joseph Poremba. Static symmetry breaking in many-sorted finite model finding. Bachelor's thesis, University of Waterloo, David R. Cheriton School of Computer Science, 2019.

[27] Jose Serna, Nancy A. Day, and Shahram Esmaeilsabzali. Dash: Declarative behavioural modelling in Alloy with control state hierarchy. *Journal of Software and Systems Modelling*, 2022.

[28] Tariq, Khadija. Linking Alloy with SMT-based finite model finding. Master's thesis, University of Waterloo, David R. Cheriton School of Computer Science, 2021.

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

[30] Amirhossein Vakili. *Temporal Logic Model Checking as Automated Theorem Proving*. PhD thesis, University of Waterloo David R. Cheriton School of Computer Science, 2016.

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

[32] Jan van Eijck. Defining (reflexive) transitive closure on finite models. https://staff.fnwi.uva.nl/d.j.n.vaneijck2/papers/08/pdfs/FinTransClosRev.pdf, June 2008.

[33] J. Zhang and H. Zhang. SEM: a system for enumerating models. In *International Joint Conference on Artificial Intelligence*, 1995.

[34] Z3 Guide. https://microsoft.github.io/z3guide/. [Online; accessed 6-July-2023].

APPENDICES

Appendix A

Results for Performance Evaluation of Transitive Closure Elimination Methods

Times considered are in seconds. TO is used to represent a test has timed out. Rows highlighted in gray are the rows considered for evaluation. Rows highlighted in green are the best method for the given problem-scope combination.

Model	Scope	Negative	Claessen	van Eijck	Squaring	Liu et al.
bit-counter	5	4.30	4.19	3.57	13.15	18.40
bit-counter	7	15.66	12.91	11.53	84.23	198.14
bit-counter	9	76.94	162.26	198.35	86.63	TO
parametric-bitcounter	5	52.61	94.16	48.46	100.36	54.13
parametric-bitcounter	7	98.26	281.44	387.11	370.71	TO
parametric-bitcounter	9	549.88	1096.05	872.15	923.87	TO
chord	5	TO	TO	TO	TO	574.62
chord	7	TO	TO	TO	TO	578.22
chord	9	TO	TO	TO	TO	577.23
digital-watch	5	247.70	490.34	1106.23	TO	TO
digital-watch	7	TO	TO	TO	TO	TO
digital-watch	9	TO	TO	TO	TO	TO
distributed-spanning-tree	5	198.77	217.03	207.37	291.79	176.55

Continued on Next Page

Continued from Previous Page						
Model	Scope	Negative	Claessen	van Eijck	Squaring	Liu et al.
distributed-spanning-tree	7	TO	TO	TO	TO	TO
distributed-spanning-tree	9	TO	TO	TO	TO	TO
ehealth	5	4.75	4.48	4.61	18.67	3.83
ehealth	7	9.95	14.99	5.64	28.45	83.11
ehealth	9	20.26	22.62	319.45	123.29	219.39
elevator	5	530.45	915.55	632.55	TO	TO
elevator	7	TO	TO	TO	TO	TO
elevator	9	TO	TO	TO	TO	TO
farmer	5	3.56	10.27	7.24	17.19	43.80
farmer	7	90.92	33.10	58.50	92.28	30.75
farmer	9	201.66	240.47	242.03	353.35	TO
musical-chairs	5	4.66	4.28	3.70	4.90	6.82
musical-chairs	7	5.94	6.39	5.56	11.60	14.04
musical-chairs	9	8.10	10.34	8.08	516.01	7.70
mutex	5	5.53	21.54	9.27	51.28	4.27
mutex	7	12.10	14.49	50.58	33.06	11.55
mutex	9	57.73	104.57	27.48	701.92	16.22
rdt	5	111.96	19.30	40.46	173.14	TO
rdt	7	TO	TO	TO	TO	TO
rdt	9	TO	TO	TO	TO	TO
ref4	5	1.81	1.90	1.82	1.95	1.86
ref4	7	2.33	2.24	2.13	5.47	7.77
ref4	9	2.72	4.31	4.63	14.46	499.43
snapshot-ui	5	1.87	1.91	1.98	3.86	1.85
snapshot-ui	7	2.23	4.60	3.02	2.33	223.78
snapshot-ui	9	7.72	4.21	7.71	51.17	TO
traffic-light	5	16.13	19.75	4.82	30.10	10.24
traffic-light	7	57.99	10.57	37.31	162.66	352.95
traffic-light	9	54.44	179.13	112.32	16.52	6.53
Total (TO=1200)		4016.72	5051.57	5693.77	7498.99	11100.44
Model	Scope	Negative	Claessen	van Eijck	Squaring	Liu et al.